KUNGL TEKNISKA HÖGSKOLAN

SWEDISH INSTITUTE OF COMPUTER SCIENCE SICS

Making reliable distributed systems in the presence of software errors

Final version (with corrections) – last update 20 November 2003

Joe Armstrong

A Dissertation submitted to
the Royal Institute of Technology
in partial fulfilment of the requirements for
the degree of Doctor of Technology
The Royal Institute of Technology
Stockholm, Sweden

December 2003

Department of Microelectronics and Information Technology

TRITA–IMIT–LECS AVH 03:09
ISSN 1651–4076
ISRN KTH/IMIT/LECS/AVH-03/09–SE

and

SICS Dissertation Series 34
ISSN 1101–1335
ISRN SICS–D–34–SE

Printed by Universitetsservice US-AB 2003

To Helen, Thomas and Claire

Abstract

The work described in this thesis is the result of a research program started in 1981 to find better ways of programming Telecom applications. These applications are large programs which despite careful testing will probably contain many errors when the program is put into service. We assume that such programs do contain errors, and investigate methods for building reliable systems despite such errors.

The research has resulted in the development of a new programming language (called Erlang), together with a design methodology, and set of libraries for building robust systems (called OTP). At the time of writing the technology described here is used in a number of major Ericsson, and Nortel products. A number of small companies have also been formed which exploit the technology.

The central problem addressed by this thesis is the problem of constructing reliable systems from programs which may themselves contain errors. Constructing such systems imposes a number of requirements on any programming language that is to be used for the construction. I discuss these language requirements, and show how they are satisfied by Erlang.

Problems can be solved in a programming language, or in the standard libraries which accompany the language. I argue how certain of the requirements necessary to build a fault-tolerant system are solved in the language, and others are solved in the standard libraries. Together these form a basis for building fault-tolerant software systems.

No theory is complete without proof that the ideas work in practice. To demonstrate that these ideas work in practice I present a number of case studies of large commercially successful products which use this technology. At the time of writing the largest of these projects is a major Ericsson

product, having over a million lines of Erlang code. This product (the AXD301) is thought to be one of the most reliable products ever made by Ericsson.

Finally, I ask if the goal of finding better ways to program Telecom applications was fulfilled—I also point to areas where I think the system could be improved.

Contents

1 Introduction

How can we program systems which behave in a reasonable manner in the presence of software errors? This is the central question that I hope to answer in this thesis. Large systems will probably always be delivered containing a number of errors in the software, nevertheless such systems are expected to behave in a reasonable manner.

To make a reliable system from faulty components places certain requirements on the system. The requirements can be satisfied, either in the programming language which is used to solve the problem, or in the standard libraries which are called by the application programs to solve the problem.

In this thesis I identify the *essential characteristics* which I believe are necessary to build fault-tolerant software systems. I also show how these characteristics are satisfied in our system.

Some of the essential characteristics are satisfied in our programming language (Erlang), others are satisfied in library modules written in Erlang. Together the language and libraries form a basis for building reliable software systems which function in an adequate manner even in the presence of programming errors.

Having said what my thesis is about, I should also say what it is not about. The thesis does not cover in detail many of the algorithms used as building blocks for construction fault-tolerant systems—it is not the algorithms themselves which are the concern of this thesis, but rather the programming language in which such algorithms are expressed. I am also

not concerned with hardware aspects of building fault-tolerant systems, nor with the software engineering aspects of fault-tolerance.

The concern is with the language, libraries and operating system requirements for software fault-tolerance. Erlang belongs to the family of *pure message passing languages*—it is a concurrent process-based language having *strong isolation* between concurrent processes. Our programming model makes extensive use of *fail-fast processes*. Such techniques are common in hardware platforms for building fault-tolerant systems but are not commonly used in software solutions. This is mainly because conventional languages do not permit different software modules to co-exist in such a way that there is no interference between modules. The commonly used threads model of programming, where resources are shared, makes it extremely difficult to isolate components from each other—errors in one component can propagate to another component and damage the internal consistency of the system.

1.1 Background

The work reported in this thesis started at the Ericsson Computer Science Laborator (CSLab) in 1981. My personal involvement started in 1985 when I joined the CSLab. The work reported here was performed in the period 1981-2003. During this time the Erlang programming language and OTP was developed by the author and his colleagues and a number of large applications were written in Erlang.

The system as we know it today is the result of the collective effort of a large number of people. Without their talents and the feedback from our users Erlang would not be what it is today. For many parts of the system it is difficult to say with precision exactly who did what and when, and exactly who had the original ideas for a particular innovation. In the acknowledgements I have tried to credit everybody as accurately as possible.

The chronology of this work is as follows:

- 1981 – The Ericsson CSLab was formed. One goal of this laboratory was *"to suggest new architectures, concepts and structures for future*

processing systems developments" [29].

- 1986 — I start work on the language that was to become Erlang, though at the time the language had no name. The language started as an experiment to add concurrent processes to Prolog—this work is described in [10]. At this stage I did not intend to design a new programming language, I was interested in how to program POTS (Plain Old Telephony Service)—at the time the best method for programming POTS appeared to be a variant of Prolog augmented with parallel processes.

- 1987 — First mention of Erlang. By 1987 the term Erlang had been coined (probably by the head of the CSLab Bjarne Däcker). By the end of the year a Prolog implementation of Erlang was available. This version of Erlang was embedded in Prolog using Prolog infix operators and did not yet have its own syntax.

  Towards the end of 1987 the first major experiment with Erlang started—a group of Ericsson engineers at Bollmora, led by Kerstin Ödling, started work on a prototyping project. They chose Erlang to prototype something called "ACS/Dunder." ACS was an architecture which was designed for implementing the Ericsson MD110 private automatic branch exchange (PABX).

  The project was to implement a number of typical PABX features in Erlang using the ACS architecture and compare the programming effort with the time it was estimated that the same job would have taken in PLEX.[1]

  Many of the ideas found in the current Erlang/OTP system can be traced back to this project.

- 1988 — Some time during 1988 it became clear that Erlang was suitable for programming Telecoms systems—so while the Bollmora group wrote applications in Erlang the CSLab group now augmented

[1] PLEX was the programming language used to program the MD110.

by Robert Virding and Mike Williams worked on improving the Erlang system.

An attempt was made to improve the efficiency of Erlang by cross compilation to the parallel logic programming language Strand. The compilation of Erlang to Strand is described in chapter 13 of [37]. Cross compilation to Strand improved the performance of Erlang by a factor of six and the project was viewed as a dismal failure.[2]

- 1989 – The ACS/Dunder project began to produce results. The Bollmora group showed that using ACS/Dunder with Erlang lead to an improvement in design efficiency of a factor of somewhere between 9 and 22 times less than the corresponding effort in PLEX. This was result was based on the experience of prototyping some 10% of the functionality of the Ericsson MD 110—these figures were hotly debated (depending on whether you believed in Erlang or not).

  The Bollmora group estimated that they would need a factor seventy in performance improvement (which we had rashly promised) in order to turn the ACS/Dunder prototype into a commercial product.

  To improve the performance of Erlang I designed the JAM machine (Joe's abstract machine). The design of the JAM machine was loosely based on the Warren abstract machine [68]. Since Erlang started as an extension to Prolog it seemed reasonable that the techniques used to efficiently implement Prolog could also be applicable to Erlang. This intuition proved correct. The JAM machine was similar to the WAM with the addition of parallel processes, message passing and failure detection and with the omission of backtracking. Compilation of pattern matching was essentially the same as in the WAM. The original JAM instruction set and details of the compilation process were published in [9].

  The design of the JAM was completed in 1989. The first implementation was an instruction set emulator written in Prolog which emulated the JAM machine. This was very inefficient and could

[2] A fact not recorded in the Strand book!.

evaluate approximately 4 reductions per second, but it was sufficient to evaluate and test the virtual machine and to allow the Erlang compiler to be written in Erlang itself.

Once the design of the JAM was complete I started a C implementation of the virtual machine—which was soon abandoned after Mike Williams read some of my C code—after which I worked on the compiler. Mike Williams wrote the virtual machine emulator and Robert Virding worked on the Erlang libraries.

- 1990 — By 1990 the JAM machine was working well and had surpassed the original goal of being seventy times faster than the original Prolog interpreter. Erlang now had its own syntax (up to now it could be regarded as a dialect of Prolog) and could be regarded as a language in its own right, rather than as a dialect of Prolog.

- 1991 — Claes Wikström added distribution to Erlang. The JAM machine was now stable and had replaced the Prolog implementation of Erlang.

- 1992 — A decision was made at Ericsson Business Systems (EBC) to develop a product based on ACS/Dunder. This product was called the *Mobility Server*—Ericsson presented Erlang developments [1, 33], at The XIV International Switching Symposium in Yokohama, Japan.

- 1993 — Ericsson starts a wholly owned subsidiary company called *Erlang Systems AB*. The purpose of this company was to market and sell Erlang to external customers and to provide training and consulting services to both internal and external customers. Support for Erlang itself was performed by the Ericsson Computer Science Laboratory. The first commercial version of Erlang was released.

- 1995 — The Ericsson AXE-N project collapsed [55]. The AXE-N project was a project to build a "next generation switch" to replace the Ericsson AXE-10. This extremely large project ran from 1987-95.

After the AXE-N project collapsed a decision was made to "restart" the project using Erlang. This project eventually resulted in the development of the AXD301 switch.

This project was on a much larger scale than any previous Erlang project. For this reason a new group was started to provide support to the AXD project. The Erlang libraries were renamed OTP (The Open Telecom Platform) and a new group was created.

- 1996 – In order to provide Erlang users with a stable software base a project called OTP (The Open Telecom Platform) was started. OTP was to be used primarily in the newly started AXD project; all existing projects were to migrate to OTP. The OTP project consolidated a number of ideas derived from experience with Erlang and in particular from a earlier set of libraries developed for use in the Mobility Server.

- 1997 – The OTP project turned into the OTP product unit which was started in order to take over formal responsibility for Erlang. Prior to that, the CSLab had formally been responsible for Erlang. I moved from the CSLab to the OTP group where I worked as the chief technical co-ordinator. During the period 1996–1997 a three-person group (myself, Magnus Fröberg and Martin Björklund) redesigned and implemented the OTP core libraries.

- 1998 – Ericsson delivered the first AXD301. The AXD301 is the subject of one of our case studies in Chapter 8. At the time of writing (2003) the AXD301 has over 1.7 million lines of Erlang code which probably makes it the largest system ever to be written in a functional style of programming.

  In February 1998 Erlang was banned for new product development within Ericsson—the main reason for the ban was that Ericsson wanted to be a consumer of software technologies rather than a producer.

  In December 1998 Erlang and the OTP libraries were released subject to an Open Source License. Since that date it has been freely

available for download from `http://www.erlang.org/`

In 1998 I left Ericsson together with a number of the original Erlang group to found a new company *Bluetail AB*—in all 15 people left Ericsson. The idea behind Bluetail was to use the Erlang technology to program products which make Internet services more reliable.

- 1999 – Bluetail produced two products written in Erlang. The *Mail Robustifier* [11] and the *Web Prioritizer*. Ericsson produced a number of Erlang products (including the AXD301 and GPRS systems).
- 2000 – Bluetail is acquired by Alteon Web Systems [3] and subsequently Alteon is acquired by Nortel Networks.
- $>$ 2001 – The Erlang/OTP technology is well established. By now there are so many projects that nobody knows the exact number. Erlang products developed by Nortel are selling for *"Hundreds of millions of kronor per year"* [51]–The Ericsson AXD301 is one of Ericsson's most successful new products and there are a dozen or so small companies using Erlang for product development.

1.2 Thesis outline

This thesis is organized into the following chapters:

- Chapter 1 introduces the main problem area that the thesis addresses, gives a background to the work in the thesis and a chronology of the work performed in the thesis together with a detailed chapter plan.
- Chapter 2 introduces an architectural model that is the basis for the later chapters in the thesis. I define what is meant by an architecture, and specify which components must be present in an architecture. I talk about the problem domain that my architecture is designed for. I talk about the underlying philosophy behind the architecture and I introduce the idea of "Concurrency Oriented Programming" (COP).

I develop the idea of COP and state the desirable properties that a programming language and system must have in order to support a concurrency oriented style of programming.

I review some previous related work, showing the similarities and differences between this prior work and the material presented in this thesis.

- Chapter 3 describes the programming language Erlang. I describe a reasonably large sub-set of the Erlang programming language, and motivate some of the design decisions made in Erlang.

- Chapter 4 gives some examples of Erlang programming techniques.

  I show how to "factor" a design into its functional and non-functional components. I show how to factor out notions of concurrency and fault-tolerance, and how to program a generic client–server model.

  I describe a technique for maintaining the illusion that "everything is an Erlang process," and give examples of how to write code which handles errors.

- Chapter 5 gets to the central question of the thesis. It is concerned with how to program systems which behave in a reasonable manner even in the presence of errors. Central to the idea of fault tolerance is the notion of an error–I describe what is meant by an error and what I mean when I talk about a "fault-tolerant" system. I describe a strategy based on the idea of "Supervision trees" which can be used for writing fault-tolerant software.

- Chapter 6 links the general principles of programming a fault-tolerant system, developed in the previous chapter, to a number of specific programming patterns developed for programming fault-tolerant systems. These programming patterns are central to the understanding of the OTP system, and of how to build fault-tolerant software in Erlang.

  I give a complete example, involving the use of a client–server model, an event-handler and a finite-state machine. These three

components are added to a supervision tree, which will monitor their progress and restart them in the event of an error.

The entire program, is packaged into a single OTP "application."

- Chapter 7 describes the OTP system. OTP stands for "Open Telecoms Platform" and is an application operating system (AOS) for programming fault-tolerant applications together with the delivery platform for the Erlang programming language. It includes a large set of libraries for implementing fault-tolerant systems, together with documentation and guides etc for understanding the system.

  In this chapter I briefly describe the OTP architecture and give details of the main components in the system.

- Chapter 8 is the acid-test of our technology. Did the ideas work in practice? In this chapter I analyse a number of large commercially successful products that make use of OTP. The intention of this chapter is to see if we have achieved our goals of programming a system which functions reliably in the presence of software errors.

  One of the projects studied in this chapter is the Ericsson AXD301, a high-performance highly-reliable ATM switch. This project is interesting in its own right, since it is one of the largest programs ever written in a functional style.

- Chapter 9 is concerned with APIs and protocols. I ask how we can specify the interfaces to modules or the interfaces between communicating components.

- In Chapter 10 I ask broader questions. Did the ideas work? Did they work well or badly? Where can things be improved? What can we look for in the future and how are we going to get there?

2 The Architectural Model

There is no standard, universally-accepted definition of the term, for software architecture is a field in its infancy, ... While there is no standard definition, there is also no shortage of them ...

The Carnegie Mellon Institute of Software Engineers

This chapter presents an architecture for building fault-tolerant systems. While everybody has a vague idea of what the word *architecture* means, there are few widely accepted definitions, which leads to many misunderstandings. The following definition captures the general idea of what is meant by a software architecture:

> *An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements, the composition of these structural and behavioural elements into progressively larger subsystems, and the architectural style that guides this organization–these elements and their interfaces, their collaborations, and their composition.*
>
> Booch, Rumbaugh, and Jacobson [19]

2.1 Definition of an architecture

At the highest level of abstraction an architecture is "a way of thinking about the world." To be useful, however, we have to turn our way of thinking about the world into a practical rulebook, and a set of procedures, that tells us how to construct a particular system using our particular way of looking at the world.

Our software architecture is characterised by descriptions of the following things:

1. A problem domain – What type of problem is the architecture designed to solve? Software architectures are not general purpose but are designed for solving specific problems. No description of an architecture is complete without describing the type of problem that is supposed to be solved using the architecture.

2. A philosophy – What is the reasoning behind the method of software construction? What are the central ideas in the architecture?

3. A set of construction guidelines – How do we program a system? We need an explicit set of construction guidelines. Our systems will be written and maintained by teams of programmers—it is important that all the programmers, and system designers, understand the system architecture, and its underlying philosophy. For practical reasons this knowledge is conveniently maintained in the form of construction guidelines. The full set of guidelines, includes sets of programming rules, and examples, course material etc.

4. A set of pre-defined components – Design by "choosing from a set of pre-defined components" is far easier than "design from scratch." The Erlang OTP libraries contain a complete set of pre-defined components (called *behaviours*) with which commonly used system components can be built. Some examples of these are the `gen_server` behaviour which can be used to build client-server systems, or the `gen_event` behaviour which can be used to build event-based pro-

grams. The pre-defined components will be discussed more fully in section 6.1.

Section 6.2.2 gives a simple example of how to program a server using the `gen_sever` behaviour.

5. A way of describing things – How can we describe the interfaces to a component? How can we describe a communication protocol between two components in our system? How can we describe the static and dynamic structure of our system? To answer these questions we will introduce a number of different, and specialised notations. Some for describing program APIs, and other notations for describing protocols, and system structure.

6. A way of configuring things – How can we start, stop, and configure, the system. How can we re-configure the system while it is in operation?

2.2 Problem domain

Our system was originally designed for building telecoms switching systems. Telecoms switching systems have demanding requirements in terms of reliability, fault-tolerance etc. Telecoms systems are expected to operate "forever," they should exhibit soft real-time behaviour, and they should behave reasonably in the presence of software and hardware errors. Däcker [30], gave ten requirements for the properties of a telecoms system:

1. The system must be able to handle very large numbers of concurrent activities.

2. Actions must be performed at a certain point in time or within a certain time.

3. Systems may be distributed over several computers.

4. The system is used to control hardware.

5. The software systems are very large.

6. The system exhibits complex functionality such as, feature interaction.

7. The systems should be in continuous operation for many years.

8. Software maintenance (reconfiguration, etc) should be performed without stopping the system.

9. There are stringent quality, and reliability requirements.

10. Fault tolerance both to hardware failures, and software errors, must be provided.

We can motivate these requirements as follows:

- Concurrency — switching systems are inherently concurrent since in a typical switch many tens of thousands of people may simultaneously interact with the switch. This implies that the system should be able to efficiently handle many tens of thousands of concurrent activities.

- Soft real-time — in a telecommunications system many operations have to be performed within a specific time. Some of these timed operations are strictly enforced, in the sense that if a given operation does not succeed within a given time interval then the entire operation will be aborted. Other operations are merely monitored with some form of timer, the operation being repeated if the timer event triggers before the operation has completed.

  Programming such systems requires manipulating many tens of thousands of timers in an efficient manner.

- Distributed — switching systems are inherently distributed, so our system should structured in such a way that it is easy to go from a single-node system to a multi-node distributed system.

- Hardware interaction — Switching systems have large amounts of peripheral hardware which must be controlled and monitored. This implies that it should be possible to write efficient device drivers, and that context switching between different device drivers should be efficient.

- Large software systems — switching systems are large, for example, the Ericsson AXE10, and the AT&T 5ESS switch, have several million lines of program code [71]. This means that our software systems must work with millions of lines of source code.

- Complex functionality — switching systems have complex functionality. Market pressure encourages the development, and deployment of systems with large numbers of complex features. Often systems are deployed before the interaction between such features is well understood. During the lifetime of a system the feature set will probably be changed and extended in many ways. Feature, and software upgrade must be performed "in place," that is, without stopping the system.

- Continuous operation — telecommunications systems are designed for many years of continuous operation. This implies that operations like software, and hardware maintenance, must be performed without stopping the system.

- Quality requirements — switching systems should run with an acceptable level of service even in the presence of errors. Telephone exchanges are expected to be extremely reliable.[1]

- Fault tolerance — switching systems should be "fault tolerant." This means that from the outset we know that faults will occur, and that we must design a software and hardware infrastructure that can deal with these faults, and provide an acceptable level of service even in the presence of faults.

[1]Typically having less than two hours of down-time in 40 years [48].

While these requirements came originally from the telecoms world they are by no means exclusive to that particular problem domain. Many modern Internet services (for example, web servers) would have a strikingly similar list of requirements.

2.3 Philosophy

How can we make a fault-tolerant software system which behaves in a reasonable manner in the presence of software errors? Answering this will take the rest of this thesis. I will start by giving a short answer which will be refined in the remainder of the thesis.

To make a fault-tolerant software system which behaves reasonably in the presence of software errors we proceed as follows:

1. We organise the software into a hierarchy of tasks that the system has to perform. Each task corresponds to the achievement of a number of goals. The software for a given task has to try and achieve the goals associated with the task.

   Tasks are ordered by complexity. The top level task is the most complex, when all the goals in the top level task can be achieved then the system should function perfectly. Lower level tasks should still allow the system to function in an acceptable manner, though it may offer a reduced level of service.

   The goals of a lower level task should be easier to achieve than the goals of a higher level task in the system.

2. We try to perform the top level task.

3. If an error is detected when trying to achieve a goal, we make an attempt to correct the error. If we cannot correct the error we immediately abort the current task and start performing a simpler task.

Programming a hierarchy of tasks needs a strong encapsulation method. We need strong encapsulation for error isolation. We want to stop pro-

gramming errors in one part of the system adversely affecting software in other parts of the system.

We need to isolate all the code that runs in order to achieve a goal in such a way that we can detect if any errors occurred when trying to achieve a goal. Also, when we are trying to simultaneously achieve multiple goals we do not want a software error occurring in one part of the system to propagate to another part of the system.

The essential problem that must be solved in making a fault-tolerant software system is therefore that of fault-isolation. Different programmers will write different modules, some modules will be correct, others will have errors. We do not want the errors in one module to adversely affect the behaviour of a module which does not have any errors.

To provide fault-isolation we use the traditional operating system notion of a process. Processes provide protection domains, so that an error in one process cannot affect the operation of other processes. Different programmers write different applications which are run in different processes; errors in one application should not have a negative influence on the other applications running in the system.

This is, of course, only true to a first approximation. Since all processes use the same CPU, and memory, processes which try to hog the CPU or which try to use excessive memory can negatively affect other processes in the system. The extent to which processes can interfere with each other depends upon the design characteristics of the operating system.

In our system processes, and concurrency, are part of the programming language and are not provided by the host operating system. This has a number of advantages over using operating system processes:

- Concurrent programs run identically on different OSs—we are not limited by how processes are implemented on any particular operating system. The only observable difference when moving between OS's, and processors should be due to different CPU speeds and memory sizes etc. All issues of synchronization, and inter-process interaction should be the same irrespective of the properties of the host operating system.

- Our language based processes are much lighter-weight than conventional OS processes. Creating a new process in our language is a highly efficient operation, some orders of magnitude faster than process creation in most operating systems[12, 14], and orders of magnitude faster than thread creation in most programming languages.

- Our system has very little need of an operating system. We make use of very few operating system services, thus it is relatively easy to port our system to specialised environments such as embedded systems.

Our applications are structured using large numbers of communicating parallel processes. We take this approach because:

1. It provides an architectural infrastructure — we can organize our system as a set of communicating processes. By enumerating all the processes in our system, and defining the message passing channels between the processes we can conveniently partition the system into a number of well-defined sub-components which can be independently implemented, and tested. This is the methodology implied by the top level of the SDL [45] system design methodology.

2. Potential efficiency — a system which is designed to be implemented as a number of independent concurrent processes can be implemented on a multi-processor or run on a distributed network of processors. Note that the efficiency is only potential, and works best when the application can be partitioned into a number of truly independent tasks. If there are strong data dependencies between the tasks this might not always be possible.

3. Fault isolation — concurrent processes with no data sharing provide a strong measure of fault isolation. A software error in a concurrent process should not influence processing in the other processes in the system.

Of these three uses of concurrency, the first two are non-essential characteristics and can be provided by some kind of in-built scheduler which provides various forms of pseudo-parallel time sharing between processes.

The third characteristic is essential for programming fault-tolerant systems. Each independent activity should be performed in a completely isolated process. Such processes should share no data, and only communicate by message passing. This is to limit the consequences of a software error.

As soon as two processes share any common resource, for example, memory or a pointer to memory, or a mutex etc the possibility exists that a software error in one of the processes will corrupt the shared resource. Since eliminating all such software errors for large software systems is an unsolved problem I think that the only realistic way to build large reliable systems is by partitioning the system into independent parallel processes, and by providing mechanisms for monitoring and restarting these processes.

2.4 Concurrency oriented programming

In our system concurrency plays a central role, so much so that I have coined the term *Concurrency Oriented Programming* to distinguish this style of programming from other programming styles.[2]

In Concurrency Oriented Programming the concurrent structure of the program should follow the concurrent structure of the application. It is particularly suited to programming applications which model or interact with the real world.

Concurrency Oriented Programming also provides the two major advantages commonly associated with object-oriented programming. These are *polymorphism* and the use of defined protocols having the same message passing interface between instances of different process types.

When we partition a problem into a number of concurrent processes we can arrange that all the processes respond to the same messages (ie

[2]Such as Object Oriented programming which models the world in terms of Objects, Functional Programming which uses functions, or Logic Programming with uses relations.

they are polymorphic,) and that they all follow the same message passing interface.

The word concurrency refers to sets of events which happen simultaneously. The real world is concurrent, and consists of a large number of events many of which happen simultaneously. At an atomic level our bodies are made up of atoms, and molecules, in simultaneous motion. At a macroscopic level the universe is populated with galaxies of stars in simultaneous motion.

When we perform a simple action, like driving a car along a freeway, we are aware of the fact that there may be several hundreds of cars within our immediate environment, yet we are able to perform the complex task of driving a car, and avoiding all these potential hazards without even thinking about it.

In the real world sequential activities are a rarity. As we walk down the street we would be very surprised to find only one thing happening, we expect to encounter many simultaneous events.

If we did not have the ability to analyze and predict the outcome of many simultaneous events we would live in great danger, and tasks like driving a car would be impossible. The fact that we can do things which require processing massive amounts of parallel information suggests that we are equipped with perceptual mechanisms which allow us to intuitively understand concurrency without consciously thinking about it.

When it comes to computer programming things suddenly become inverted. Programming a sequential chain of activities is viewed the norm , and in some sense is thought of as being easy, whereas programming collections of concurrent activities is avoided as much as possible, and is generally perceived as being difficult.

I believe that this is due to the poor support which is provided for concurrency in virtually all conventional programming languages. The vast majority of programming languages are essentially sequential; any concurrency in the language is provided by the underlying operating system, and not by the programming language.

In this thesis I present a view of the world where concurrency is provided by the programming language, and not by the underlying operating system. Languages which have good support for concurrency I call *Concur-*

rency Oriented Languages, or COPLs for short.

2.4.1 Programming by observing the real world

We often want to write programs that model the world or interact with the world. Writing such a program in a COPL is easy. Firstly, we perform an analysis which is a three-step process:

1. We identify all the truly concurrent activities in our real world activity.

2. We identify all message channels between the concurrent activities.

3. We write down all the messages which can flow on the different message channels.

Now we write the program. The structure of the program should exactly follow the structure of the problem. Each real world concurrent activity should be mapped onto exactly one concurrent process in our programming language. If there is a 1:1 mapping of the problem onto the program we say that the program is isomorphic to the problem.

It is extremely important that the mapping is exactly 1:1. The reason for this is that it minimizes the conceptual gap between the problem and the solution. If this mapping is not 1:1 the program will quickly degenerate, and become difficult to understand. This degeneration is often observed when non-CO languages are used to solve concurrent problems. Often the only way to get the program to work is to force several independent activities to be controlled by the same language thread or process. This leads to a inevitable loss of clarity, and makes the programs subject to complex and irreproducible interference errors.

In performing our analysis of the problem we must choose an appropriate granularity for our model. For example, if we were writing an instant messaging system, we might choose to use one process per user and not one process for every atom in the user's body.

2.4.2 Characteristics of a COPL

COPLs are characterised by the following six properties:

1. COPLs must support processes. A process can be thought of as a self-contained virtual machine.

2. Several processes operating on the same machine must be strongly isolated. A fault in one processe should not adversely effect another process, unless such interaction is explicitly programmed.

3. Each process must be identified by a unique unforgeable identifier. We will call this the Pid of the process.

4. There should be no shared state between processes. Processes interact by sending messages. If you know the Pid of a process then you can send a message to the process.

5. Message passing is assumed to be unreliable with no guarantee of delivery.

6. It should be possible for one process to detect failure in another process. We should also know the reason for failure.

Note that COPLs must provide true concurrency, thus objects represented as processes are truly concurrent, and messages between processes are true asynchronous messages, unlike the disguised remote procedure calls found in many object-oriented languages.

Note also that the reason for failure may not always be correct. For example, in a distributed system, we might receive a message informing us that a process has died, when in fact a network error has occurred.

2.4.3 Process isolation

The notion of *isolation* is central to understanding COP, and to the construction of fault-tolerant software. *Two processes operating on the same*

machine must be as independent as if they ran on physically separated machines.

Indeed the ideal architecture to run a CO program on would be a machine which assigned one new physical processor per software process. Until this ideal is reached we will have to live with the fact that multiple processes will run on the same machine. We should still think of them as if they ran on physically separated machine.

Isolation has several consequences:

1. Processes have "share nothing" semantics. This is obvious since they are imagined to run on physically separated machines.

2. Message passing is the only way to pass data between processes. Again since nothing is shared this is the only means possible to exchange data.

3. Isolation implies that message passing is asynchronous. If process communication is synchronous then a software error in the receiver of a message could indefinitely block the sender of the message destroying the property of isolation.

4. Since nothing is shared, everything necessary to perform a distributed computation must be copied. Since nothing is shared, and the only way to communicate between processes is by message passing, then we will never know if our messages arrive (remember we said that message passing is inherently unreliable.) The only way to know if a message has been correctly sent is to send a confirmation message back.

Programming a system of processes subject to the above rules may appear at first sight to be difficult—after all most concurrency extensions to sequential programming languages provide facilities for almost exactly the opposite providing things like locks, and semaphores, and provision for shared data, and reliable message passing. Fortunately, the opposite turns out to be true—programming such a system turns out to be surprisingly

easy, and the programs you write can be made scalable, and fault-tolerant, with very little effort.

Because all our processes are required to be complete isolated adding more processes cannot affect the original system. The software must have been written so as to handle collections of isolated processes, so adding a few more processors is usually accomplished without any major changes to the application software.

Since we made no assumptions about reliable message passing, and must write our application so that it works in the presence of unreliable message passing it should indeed work in the presence of message passing errors. The initial effort involved will reward us when we try to scale up our systems.

2.4.4 Names of processes

We require that the names of processes are unforgeable. This means that it should be impossible to guess the name of a process, and thereby interact with that process. We will assume that processes know their own names, and that processes which create other processes know the names of the processes which they have created. In other words, a parent process knows the names of its children.

In order to write COPLs we will need mechanisms for finding out the names of the processes involved. Remember, if we know the name of a process, we can send a message to that process.

System security is intimately connected with the idea of knowing the name of a process. If we do not know the name of a process we cannot interact with it in any way, thus the system is secure. Once the names of processes become widely know the system becomes less secure. We call the process of revealing names to other processes in a controlled manner the *name distribution problem*— the key to security lies in the name distribution problem. When we reveal a Pid to another process we will say that we have published the name of the process. If a name is never published there are no security problems.

Thus knowing the name of a process is the key element of security. Since names are unforgeable the system is secure only if we can limit the

knowledge of the names of the processes to trusted processes.

In many primitive religions it was believed that humans had powers over spirits if they could command them by their real names. Knowing the real name of a spirit gave you power over the spirit, and using this name you could command the spirit to do various things for you. COPLs use the same idea.

2.4.5 Message passing

Message passing obeys the following rules:

1. Message passing is assumed to be atomic which means that a message is either delivered in its entirety or not at all.

2. Message passing between a pair of processes is assumed to be ordered meaning that if a sequence of messages is sent and received between any pair of processes then the messages will be received in the same order they were sent.

3. Messages should not contain pointers to data structures contained within processes—they should only contain constants and/or Pids.

Note that point two is a design decision, and does not reflect any underlying semantics in the network used to transmit messages. The underlying network might reorder the messages, but between any pair of processes these messages can be buffered, and re-assembled into the correct order before delivery. This assumption makes programming message passing applications much easier than if we had to always allow for out of order messages.

We say that such message passing has *send and pray semantics.* We *send* the message and *pray* that it arrives. Confirmation that a message has arrived can be achieved by returning a confirmation message (sometimes called round-trip confirmation.) Interestingly many programmers only believe in round-trip confirmation, and use it even if the underlying transport layers are supposed to provide reliable data transport, and even if such checks are supposedly irrelevant.

Message passing is also used for *synchronisation*. Suppose we wish to synchronise two processes `A`, and `B`. If `A` sends a message to `B` then `B` can only receive this message at some point in time *after* `A` has sent the message. This is known as *casual ordering* in distributed systems theory. In COPLs all interprocess synchronisation is based on this simple idea.

2.4.6 Protocols

Isolation of components, and message passing between components, is architecturally sufficient for protecting a system from the consequences of a software error, but it is not sufficient to specify the behaviour of a system, nor, in the event of some kind of failure to determine which component has failed.

Up to now we have assumed that failure is a property of a single component, a single component will either do what it is supposed to do or fail as soon as possible. It might happen, however, that no components are observed to fail, and yet the system still does not work as expected.

To complete our programming model, we add therefore one more thing. Not only do we need completely isolated components that communicate only by message passing, but also we need to specify the communication protocols that are used between each pair of components that communicate with each other.

By specifying the communication protocol that should be obeyed between two components we can easily find out if either of the components involved has violated the protocol. Guaranteeing that the protocol is enforced should be done by static analysis, if possible, or failing this by compiling run-time checks into the code.

2.4.7 COP and programmer teams

Building a large system involves the work of many programmers, sometimes many hundreds of programmers are involved. To organise their work these programmers are organised into smaller groups or teams. Each group is responsible for one or more logical component in the system. On a day-to-day basis, the groups communicate by message-passing (e-mail

or phone) but do not regularly meet. In some cases the groups work in different countries, and never meet. It is amusing to note that not only is the organisation of a software system into isolated components which communicate by pure message passing desirable for a number of reasons, but also that it is the way that large programming groups are organised.

2.5 System requirements

To support a CO style of programming, and to make a system that can satisfy the requirements of a telecoms system we arrive at a set of requirements for the essential characteristics of a system. These requirements are for the system as a whole–here I am not interested in whether these requirements are satisfied in a programming language or in the libraries, and construction methods, which accompany the language.

There are six essential requirements on the underlying operating system, and programming languages.

R1. Concurrency – Our system must support concurrency. The computational effort needed to create or destroy a concurrent process should be very small, and there should be no penalty for creating large numbers of concurrent processes.

R2. Error encapsulation – Errors occurring in one process must not be able to damage other processes in the system.

R3. Fault detection – It must be possible to detect exceptions both locally (in the processes where the exception occurred,) and remotely (we should be able to detect that an exception has occurred in a non-local process).

R4. Fault identification – We should be able to identify why an exception occurred.

R5. Code upgrade – there should exist mechanisms to change code as it is executing, and without stopping the system.

R6. Stable storage — we need to store data in a manner which survives a system crash.

It is also important that systems satisfying the above requirements are efficiently implemented—concurrency is not much use if we cannot reliably create many tens of thousands of processes. Fault identification is not much use if it does not contain enough information to allow us to correct the error at a later date.

Satisfying the above requirements can be done in a number of different ways. Concurrency, for example, can be provided as a language primitive (as, for example, in Erlang), or in the operating system (for example, Unix). Languages like C or Java which are not concurrent can make use of operating system primitives which gives the user the illusion that they are concurrent; indeed concurrent programs can be written in languages which are not themselves concurrent.

2.6 Language requirements

The programming language which we use to program the system must have:

- Encapsulation primitives — there must be a number of mechanisms for limiting the consequences of an error. It should be possible to isolate processes so that they cannot damage each other.
- Concurrency — the language must support a lightweight mechanism to create parallel process, and to send messages between the processes. Context switching between process, and message passing, should be efficient. Concurrent processes must also time-share the CPU in some reasonable manner, so that CPU bound processes do not monopolise the CPU, and prevent progress of other processes which are "ready to run."
- Fault detection primitives — which allow one process to observe another process, and to detect if the observed process has terminated for any reason.

- Location transparency – If we know the Pid of a process then we should be able to send a message to the process.

- Dynamic code upgrade – It should be possible to dynamically change code in a running system. Note that since many processes will be running the same code, we need a mechanism to allow existing processes to run "old" code, and for "new" processes to run the modified code at the same time.

Not only should the language satisfy these requirements, but it should also satisfy them in a reasonably efficient manner. When we program we do not want to limit our freedom of expression by "counting processes" etc, nor do we want to worry about what will happen if a process inadvertently tries to monopolise the CPU.

The maximum number of processes in the system should be sufficiently large that for programming purposes we do not have to consider this maximum number a limiting factor. We might need, for example, to create of the order of one hundred thousand processes in order to make a switching system which maintains ten thousand simultaneous user sessions.[3]

This mix of features is needed to simplify applications programming. Mapping the semantics of a distributed set of communicating components onto an Erlang program is greatly simplified if we can map the concurrent structure of the problem in a 1:1 manner onto the process structure of the application program which solves the problem.

2.7 Library requirements

Language is not everything–a number of things are provided in the accompanying system libraries. The *essential* set of libraries routines must provide:

- Stable storage – this is storage which survives a crash.

[3] Assuming 10 processes per session.

- Device drivers — these must provide a mechanism for communication with the outside world.
- Code upgrade — this allows us to upgrade code in a running system.
- Infrastructure — for starting, and stopping the system, logging errors etc.

Observe that our library routines, which are mostly written in Erlang, provide most of the services which are conventionally provided by an operating system.

Since Erlang process are isolated from each other, and communicate only by message passing, they behave very much like operating system processes which communicate through pipes or sockets.

Many of the features which are conventionally provided by an operating system have moved from the operating system into the programming language. The remaining operating system only provides a primitive set of device drivers.

2.8 Application libraries

Stable storage etc is not provided as a language primitive in Erlang, but is provided in the basic Erlang libraries. Having such libraries is a precondition for writing any complex application software. Complex applications need much higher-level abstractions than storage etc. To build such applications we need pre-packaged software entities to help us program things like client-server solutions etc.

The OTP libraries provide us with a complete set of design patterns (called behaviours) for building fault-tolerant applications. In this thesis I will talk about a minimal set of behaviours, which can be used for building fault-tolerant applications. These are:

- `supervisor` — a supervision model.
- `gen_server` — a behaviour for implementing client-server applications.

- `gen_event` – a behaviour used for implementing event handling software.

- `gen_fsm` – a behaviour used for implementing finite state machines.

Of these, the central component that is used for programming fault-tolerant applications is the supervision model.

2.9 Construction guidelines

In addition to explaining a general philosophy of programming fault-tolerant applications, we need more specific guidelines that apply to the programming languages that we wish to use to program our applications. We also need example programs, and examples of how to use the library routines.

The open source Erlang release contains such guidelines which have been used as the basis for systems with millions of lines of Erlang code. Appendix B reproduces the programming guidelines, which can be found in the Erlang open source release. This thesis contains additional guidelines, which are organized as follows:

- The overall philosophy of our architecture is described in this chapter.

- The notion of an error is discussed in several places. Sections 5.3, and 4.3 describe what is meant by an error; section 4.4 gives advice on the correct programing style to use when programming for errors in Erlang.

- Examples of how to program simple components can be found in Chapter 4, and examples of how to use the OTP behaviours in chapter 6.

2.10 Related work

The inability to isolate software components from each other is the main reason why many popular programming languages cannot be used for making robust system software.

> *It is essential for security to be able to isolate mistrusting programs from one another, and to protect the host platform from such programs. Isolation is difficult in object-oriented systems because objects can easily become aliased.*[4]—Bryce [21]

Bryce goes on to say that object aliasing is difficult if not impossible to detect in practice, and recommends the use of *protection domains* (akin to OS processes) to solve this problem.

In a paper on Java Czajkowski, and Daynès, from Sun Microsystems, write:

> *The only safe way to execute multiple applications, written in the Java programming language, on the same computer is to use a separate JVM for each of them, and to execute each JVM in a separate OS process. This introduces various inefficiencies in resource utilization, which downgrades performance, scalability, and application startup time. The benefits the language can offer are thus reduced mainly to portability and improved programmer productivity. Granted these are important, but the full potential of language-provided safety is not realized. Instead there exists a curious distinction between "language safety," and "real safety".* — [28]

In this paper they introduce the MVM (an extension to the JVM) where their goal is:

> *... to turn the JVM into an execution environment akin to an OS. In particular, the abstraction of a process, offered by*

[4] An aliased object is one where at least two other objects hold a reference to it.

> *modern OSes, is the role model in terms of features; isolation from other computations, resources accountability and control, and ease of termination and resource reclamation.*

To achieve this they conclude that:

> *... tasks cannot directly share objects, and that the only way for tasks to communicate is to use standard, copying communication mechanisms, ...*

These conclusions are not new. Very similar conclusions were arrived at some two decades earlier by Jim Gray who described the architecture of the Tandem Computer in his highly readable paper *Why do computers stop and what can be done about it.* He says:

> *As with hardware, the key to software fault-tolerance is to hierarchically decompose large systems into modules, each module being a unit of service and a unit of failure. A failure of a module does not propagate beyond the module.*
>
> *...*
>
> *The process achieves fault containment by sharing no state with other processes; its only contact with other processes is via messages carried by a kernel message system.* — [38]

Language which support this style of programming (parallel processes, no shared data, pure message passing) are what Andrews and Schneider [4] refer to as a "Message oriented languages." The language with the delightful name PLITS[5] (1978) [35] is probably the first example of such a programming language:

> *The fundamental design decision in the implementation of RIG[6] was to allow a strict message discipline with no shared*

[5]Programming language in the sky.

[6]RIG was a small system written in PLITS.

> *data structures. All communication between user and server messages is through messages which are routed by the Aleph kernel. This message discipline has proved to be very flexible and reliable.* — [35]

Turning away from language for a while, we ask what properties should an individual process have?

Schneider [60, 59] answered this question by giving three properties that he thought a hardware system should have in order to be suitable for programming a fault-tolerant system. These properties Schneider called:

1. *Halt on failure* — in the event of an error a processor should halt instead of performing a possibly erroneous operation.

2. *Failure status property* — when a processor fails, other processors in the system must be informed. The reason for failure must be communicated.

3. *Stable Storage Property* — the storage of a processor should be partitioned into stable storage (which survives a processor crash,) and volatile storage which is lost if a processor crashes.

A processor having these properties Schneider called a *fail-stop processor*. The idea is that if a failure[7] occurs it is pointless to continue. The process should halt instead of continuing and possibly causing more damage. In a fault-stop processor, state is stored in either volatile or stable memory. When the processor crashes all data in volatile memory is lost, but all state that was stored in stable storage should be available after the crash.

A remarkably similar idea can be found in [38] where Gray talks about "fail-fast" processes.

> *The process approach to fault isolation advocates that the process software be fail-fast, it should either function correctly or it should detect the fault, signal failure and stop operating.*

[7] Here Schneider considers a failure as an error which cannot be corrected.

> *Processes are made fail-fast by defensive programming. They check all their inputs, intermediate results and data structures as a matter of course. If any error is detected, they signal a failure and stop. In the terminology of [Cristian], fail-fast software has small fault detection latency.* — [38]

Both Schneider, and Gray, have the same essential idea; one is talking about hardware, the other about software but the underlying principles are the same.

Renzel argued that it is important that processes fail as soon as possible after an uncorrectable error has occurred:

> *A fault in a software system can cause one or more errors. The latency time which is the interval between the existence of the fault and the occurrence of the error can be very high, which complicates the backwards analysis of an error ...*
>
> *For an effective error handling we must detect errors and failures as early as possible* — [58]

Bearing in mind these arguments, and our original requirements I advocate a system with the following properties:

1. Processes are the units of error encapsulation — errors occurring in a process will not affect other processes in the system. We call this property *strong isolation.*

2. Processes do what they are supposed to do or fail as soon as possible.

3. Failure, and the reason for failure, can be detected by remote processes.

4. Processes share no state, but communicate by message passing.

A language and system with such properties, has the necessary preconditions for writing fault-tolerant software. Later in this thesis we will see how these properties are satisfied by Erlang, and the Erlang libraries.

Many of the ideas in this thesis are not new—the fundamental principles for making a fault-tolerant system are described in Gray's [38] paper.

Many of the features of the Tandem computer bear a striking similarity to the design principles in the OTP system, and to the fundamental principles of Concurrency Oriented Programming which where discussed earlier.

Here are two quotes from the paper, firstly the design principles on page 15 of [38].

> *The keys to this software fault-tolerance are:*
>
> - *Software modularity through processes, and messages.*
> - *Fault containment through fail-fast software modules.*
> - *Process-pairs to tolerant hardware, and transient software faults.*
> - *Transaction mechanisms to provide data, and message integrity.*
> - *Transaction mechanisms combined with process-pairs to ease exception handling, and tolerate software faults.*
>
> *Software modularity through processes and messages. As with hardware, the key to software fault-tolerance is to hierarchically decompose large systems into modules, each module being a unit of service and a unit of failure. A failure of a module does not propagate beyond the module.*
>
> *There is considerable controversy about how to modularize software. Starting with Burroughs' Espol and continuing through languages like Mesa and Ada, compiler writers have assumed perfect hardware and contended that they can provide good isolation through static compile-time type checking. In contrast, operating systems designers have advocated run-time checking combined with the process as the unit of protection and failure.*

> *Although compiler checking and exception handling provided by programming languages are real assets, history seems to have favored the run-time checks plus the process approach to fault-containment. It has the virtue of simplicity—if a process or its processor misbehaves, stop it. The process provides a clean unit of modularity, service, fault containment and failure.*
>
> *Fault containment through fail-fast software modules.*
>
> *The process achieves fault containment by sharing no state with other processes; its only contact with other processes is via messages carried by a kernel message system.* — [38]

If we compare this to our current Erlang system we see many striking similarities. There are certain difference—in Erlang "defensive programming" is not recommended since the compiler adds the necessary checks to make this style of programming unnecessary. Gray's "transaction mechanism" is provides by the mnesia data base.[8] The containment and processing of errors is managed by the "supervision tree" behaviours in the OTP libraries.

The idea of "fail-fast" modules is mirrored in our guidelines for programming where we say that processes should only do when they are supposed to do according to the specification, otherwise they should crash. The supervision hierarchies in our system correspond to the hierarchies of modules that Gray refers to. This idea can also be found in the work of Candea and Fox [22] who talked about "crash-only software"—they argue that allowing components to crash and then restart leads to a simpler fault model and more reliable code.

More modern work with object-oriented systems has also recognised the importance of isolating software components from each other. In [21] Bryce and Razafimahefa argue that is is essential to isolate programs from one another, and from the programs which run in the host operating system. This, they consider, is the *essential* characteristic that any object system must have. As they point out in their paper, this is a difficult problem in an object-oriented context.

[8] Written in Erlang.

3 Erlang

This chapter introduces Erlang. The treatment of the language is not intended to be complete. For fuller treatment the reader is referred to [5]. Developments to Erlang since [5] can be found in the OTP documentation [34]. A more formal treatment of Erlang can be found in the Erlang Specification [17] and in the core Erlang specification [23].

Erlang belongs to the class of *Message-oriented languages* [4] — message oriented languages provide concurrency in the form of parallel processes. There are no shared objects in a message-oriented language. Instead all interaction between processes is achieved by sending and receiving messages.

In this chapter, I present a subset of the language which provides enough detail to understand the Erlang examples in this thesis.

3.1 Overview

The Erlang view of the world can be summarized in the following statements:

- Everything is a process.
- Processes are strongly isolated.
- Process creation and destruction is a lightweight operation.

- Message passing is the only way for processes to interact.
- Processes have unique names.
- If you know the name of a process you can send it a message.
- Processes share no resources.
- Error handling is non-local.
- Processes do what they are supposed to do or fail.

The use of processes as the basic unit of abstraction is motivated by the desire to make a language which is suitable for writing large fault-tolerant software systems. The fundamental problem which must be solved in writing such software is that of limiting the consequences of an error– the process abstraction provides an abstraction boundary which stops the propagation of errors.

It is, for example, precisely this inability to limit the consequences of errors that makes Java unsuitable for programming "safe" (sic) applications (see page 32 for further discussion of this point).

If processes are truly isolated (which they must be to limit the consequences of an error) then most of the other properties of a process, like, for example, that the only way for processes to interact is by message passing, etc, follow as a natural consequence of this isolation.

The statement about error handling is perhaps less obvious. When we make a fault-tolerant system we need at least *two* physically separated computers. Using a single computer will not work, if it crashes, all is lost. The simplest fault-tolerant system we can imagine has exactly two computers, if one computer crashes, then the other computer should take over what the first computer was doing. In this simple situation even the software for fault-recovery must be non-local; the error occurs on the first machine, but is corrected by software running on the second machine.

The Erlang view of the world is that "everything is a process", when we model our physical machines as processes we retain the idea that error handling should be non-local. Actually, this is a modified truth, remote

error handling only occurs if a local attempt to fix the error fails. In the event of an exception a local process may be able to detect and correct the fault which caused the exception, in which case as far as any other process in the system is concerned, no error has occurred.

Viewed as a concurrent language, Erlang is very simple. Since there are no shared data structures, no monitors or synchronised methods etc there is very little to learn. The bulk of the language, and possible the least interesting part of the language is the sequential subset of the language. This sequential subset can be characterised as a dynamically typed, strict functional programming language, which is largely free from side-effects. In the sequential subset there are a few operations with side-effects, but they are virtually never needed.

The remainder of this chapter deals firstly with the sequential subset of the language. This is followed with sections on concurrent and distributed programming and error handling. Finally I describe a type notation for specifying Erlang data and function types.

To jump start the description, I start with an example of sequential Erlang code.

3.2 Example

Figure 3.1 has a simple Erlang program. The program has the following structure:

1. The program starts with a *module* definition (line 1) followed by *export* and *input* declarations and then by a number of *functions*.

2. The export declaration (line 2) says that the function `areas/1` is to be exported from this module. The notation `areas/1` means the function called `areas` which has one argument. The only functions which can be called from outside the module are those which are contained in the export list.

3. The import declaration in line 3 says that the function `map/2` can be found in the module `lists`.

```
1  -module(math).
2  -export([areas/1]).
3  -import(lists, [map/2]).
4
5  areas(L) ->
6      lists:sum(
7                 map(
8                     fun(I) -> area(I) end,
9                     L)).
10
11 area({square, X}) ->
12     X*X;
13 area({rectangle,X,Y}) ->
14     X*Y.
```

Figure 3.1: An Erlang module

4. Lines 5 to 14 have two function definitions.

5. Line 6 is a call to the function `sum` in the module `lists`.

6. Lines 7 to 9 are a call to the function `map/2` in the module `lists`. Note the difference between this call to `sum` and the call to `map` - both these functions are in the same module; one call uses a *fully qualified name* (that is, `lists:sum`) whereas the other call uses an abbreviated call sequence (that is `map(...)` instead of `lists:map(...)`). The difference is accounted for by the import declaration in line 3, which says that the function `map/2` is to be found in the module `lists`.

7. Line 8 creates a *fun* which is the first argument to `map`.

8. Lines 11 to 14 contain the function `area/1`. This function has two *clauses*. The first clause is in lines 11 to 12, the second in lines 13 to 14, the clauses are separated by a semi-colon.

9. Each clause has a *head* and a *body*. The head and body are separated from each other by a "`->`" symbol.

10. A function head consists of a *pattern* in each argument position and a possible guard (See Section 3.3.4). In line 13 the pattern is `{rectangle,X,Y}`. In this pattern the curly bracket denote a *tuple*. The first argument of the tuple is an *atom* (namely "`rectangle`") and the second and third arguments are *variables*. Variables start with capital letters, atoms start with small letters.

To run this program we start an Erlang shell compile the program and enter some requests to evaluate functions, as shown in Figure 3.2. In this figure all user input is underlined. The Erlang shell prompt is the character "`>`" meaning that the system is waiting for input.

- Line 1 in figure 3.2 starts an Erlang shell.

- Line 5 compiles the module `math`.

```
1  $ erl
2  Erlang (BEAM) emulator version 5.1 [source]
3
4  Eshell V5.1  (abort with ^G)
5  1> c(math).
6  ok,math
7  2> math:areas([{rectangle,12,4},{square,6}]).
8  84
9  3> math:area({square,10}).
10 ** exited: {undef,[{math,area,[{square,10}]},
11                   {erl_eval,expr,3},
12                   {erl_eval,exprs,4},
13                   {shell,eval_loop,2}]} **
```

Figure 3.2: Compiling and running a program in the shell

- Line 7 requests a function evaluation, the shell accepts the request, evaluates the function and prints the result in line 8.
- In line 9 we try to evaluate a function which was not exported from the module `math`. An exception is generated and printed (lines 10 to 13).

3.3 Sequential Erlang

3.3.1 Data structures

Erlang has eight primitive data types:[1]

- **Integers** — integers are written as sequences of decimal digits, for example, `12`, `12375` and `-23427` are integers. Integer arithmetic is

[1] Also called constants.

exact and of unlimited precision.[2]

- **Atoms** – atoms are used within a program to denote distinguished values. They are written as strings of consecutive alphanumeric characters, the first character being a small letter. Atoms can obtain any character if they are enclosed within single quotes and an escape convention exists which allows any character to be used within an atom.

- **Floats** – floating point numbers are represented as IEEE 754 [43] 64 bit floating point numbers. Real numbers in the range $\pm 10^{308}$ can be represented by an Erlang float.

- **References** – references are globally unique symbols whose only property is that they can be compared for equality. They are created by evaluating the Erlang primitive `make_ref()`.

- **Binaries** – a binary is a sequence of bytes. Binaries provide a space-efficient way of storing binary data. Erlang primitives exist for composing and decomposing binaries and for efficient input/output of binaries. For a full treatment of binaries see [34].

- **Pids** – Pid is short for *Process Identifier*–a Pid is created by the Erlang primitive `spawn(...)` Pids are references to Erlang processes.

- **Ports** – ports are used to communicate with the external world. Ports are created with the BIF[3] `open_port`. Messages can be sent to and received from ports, but these message must obey the so-called "port protocol."

- **Funs** – Funs are function closures.[4] Funs are created by expressions of the form: `fun(...) -> ... end`.

And two compound data types:

[2]The precision of integers is only limited by available memory.

[3]BIF is short for Built In Function.

[4]Called lambda expressions in other languages.

- **Tuples** – tuples are containers for a fixed number of Erlang data types. The syntax `{D1,D2,...,Dn}` denotes a tuple whose arguments are `D1`, `D2`, ... `Dn`. The arguments can be primitive data types or compound data types. The elements of a tuple can be accessed in constant time.

- **Lists** – lists are containers for a variable number of Erlang data types. The syntax `[Dh|Dt]` denotes a list whose first element is `Dh`, and whose remaining elements are the list `Dt`. The syntax `[]` denotes an empty list.

  The syntax `[D1,D2,..,Dn]` is short for `[D1|[D2|..|[Dn|[]]]]`. The first element of a list can be accessed in constant time. The first element of a list is called the *head* of the list. The remainder of a list when its head has been removed is called the *tail* of the list.

Two forms of syntactic sugar are provided:

- **Strings** – strings are written as doubly quoted lists of characters, this is syntactic sugar for a list of the integer ASCII codes for the characters in the string, thus for example, the string `"cat"` is shorthand for `[97,99,116]`.

- **Records** – records provide a convenient way for associating a tag with each of the elements in a tuple. This allows us to refer to an element of a tuple by name and not by position. A pre-compiler takes the record definition and replaces it with the appropriate tuple reference.

3.3.2 Variables

Variables in Erlang are sequences of characters starting with a upper case letter and followed by a sequence of letters or characters or the "_" character.

Variables in Erlang are either *unbound,* meaning they have no value, or *bound,* meaning that they have a value. Once a variable has been bound

the value can never be changed. Such variables are called *single assignment variables*. Since variable values cannot ever be changed the programmer must create a new variable every time they want to simulate the effect of a destructive assignment.

Thus, for example, the Erlang equivalent of the `C` expression:

```
x = 5;
x = x + 10;
```

is written:

```
X = 5;
X1 = X + 10;
```

Where we invent a new variable `X1` since we cannot change the value of `X`.

3.3.3 Terms and patterns

A *Ground term* is defined recursively as either a primitive data type, or a tuple of ground terms or a list of ground terms.

A *Pattern* is defined recursively as either a primitive data type or a variable or a tuple of patterns or a list of patterns.

A *Primitive pattern* is a pattern where all the variables are different.

Pattern matching is the act of comparing a pattern with a ground term. If the pattern is a primitive pattern and the ground terms are of the same shape, and if the constants occurring in the pattern occur in the ground term in the same places as in the pattern then the match will succeed, otherwise it will fail. Any variables occurring in the pattern will be bound to the corresponding data items at the same positions in the term. This process is called *unification*.

More formally if `P` is a primitive pattern and `T` is a term, then we say that `P` matches `T` iff:

- If `P` is a list with head `Ph` and tail `Pt` and `T` is a list with `Th` and tail `Tt` then `Ph` must match `Th` and `Pt` must match `Tt`.

- If `P` is a tuple with elements `{P1,P2,...,Pn}` and `T` is a tuple with elements `{T1,T2,...,Tn}` then `P1` must match `T1`, `P2` must match `T2` and so on.

- If `P` is a constant then `T` must be the same constant.

- If `P` is a free variable `V` then `V` is bound to `T`.

Here are some examples:

The pattern `{P,abcd}` matches the term `{123,abcd}` creating the binding `P` $\mapsto$ `123`.

The pattern `[H|T]` matches the term `"cat"` creating the bindings `H` $\mapsto$ `99` and `T` $\mapsto$ `[79,116]`.

The Pattern `{abc,123}` does not match the term `{abc,124}`.

3.3.4 Guards

Guards are expressions involving only predicates. They are written immediately after primitive patterns and introduced with the keyword `when`. For example we call the program fragment:

```
{P,abc,123} when P == G
```

a guarded pattern.

Guards are written as comma-separated sequences of guard tests, where each guard test is of the form:

```
T1 Binop T2
```

where `T1` and `T2` are ground terms.

The available binary operators are:

Operator	Meaning
`X > Y`	`X` is greater than `Y`
`X < Y`	`X` is less than `Y`
`X =< Y`	`X` is equal to or less than `Y`
`X >= Y`	`X` is greater than or equal to `Y`
`X == Y`	`X` is equal to `Y`
`X /= Y`	`X` is not equal to `Y`
`X =:= Y`	`X` is equal to `Y`
`X =/= Y`	`X` is not equal to `Y`

When a guard is used as an expression, it will always evaluate to one of the atoms `true` or `false`. If the guard evaluates to `true` we say that the evaluation *succeeded* otherwise it *failed*.

3.3.5 Extended pattern matching

In a primitive pattern all variables must be different. An extended pattern has the same syntax as a primitive pattern except that all the variables are not required to be different.

To perform pattern matching with extended patterns we first convert the extended pattern to a primitive pattern and guard and then match the primitive pattern.

If the variable `X` is repeated in the pattern `N` times then rename the second and all subsequent occurrences of `X` with a fresh variables.[5] `F1`, `F2` etc. For each fresh variable add a predicate `Fi == X` to the guard.

Using these rules

```
{X,a,X,[B|X]}
```

is transformed into

```
{X,a,F1,[B|F2]} when F1==X, F2==X
```

Finally, the pattern variable '_' is taken to mean the "anonymous variable." The anonymous variable matches any term and no variable binding is created.

[5] A fresh variable is one that does not occur anywhere in the current lexical context.

3.3.6 Functions

Functions obey the following rules:

1. A function is composed of one or more *clauses* which are separated by semi-colons.

2. A clause has a *head* followed by a separator `->` followed by a *body*.

3. A function head is composed of an atom followed by a parenthesized set of patterns followed by an optional *guard*. The guard, if present, is introduced by the `when` keyword.

4. A function body consists of a sequence of comma-separated *expressions*.

Or,

```
FunctionName(P11,...,P1N) when G11,...,G1N ->
    Body1;
FunctionName(P21,...,P2N) when G11,...,G1N ->
    Body2;
...
FunctionName(PK1, PK2, ..., PKN) ->
    BodyK.
```

Where `P11, ..., PKN` are the extended patterns described in the previous section.

Here are two examples:

```
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

member(H, [H|T]) -> true;
member(H, [_|T]  -> member(H, T);
member(H, [])    -> false.
```

Function execution is as follows:

To evaluate `Fun(Arg1,Arg2,...,ArgN)` we first search for a definition of the function. The corresponding definition is taken to be the first function whose patterns in the head of the clause match the arguments `Arg1..Argn` in the function call. If the pattern match succeeds and if any associated guard test succeeds then the body of the clause is evaluated. All free variables occurring in the patterns in the head of the clause have values that were obtained as a result of matching the pattern in the clause head with the actual arguments provided in the call. As an example we evaluate the expression `member(dog,[cat,man,dog,ape])` showing all steps taken. We assume the following definition of `member`:

```
member(H, [H1|_]) when H == H1 -> true;
member(H, [_|T]  -> member(H, T);
member(H, [])    -> false.
```

1. Evaluate `member(dog, [cat,man,dog,ape])`

2. The first clause matches with bindings {`H` $\mapsto$ `dog`, `H1` $\mapsto$ `cat`}. The guard test then fails.

3. The second clause matches with {`H` $\mapsto$ `dog`, `T` $\mapsto$ `[man,dog,ape]`}, there is no guard test so the system evaluates `member(H,T)` with the current bindings of `H` and `T`.

4. Evaluate `member(dog, [man,dog,ape])`

5. As before. This time the second clause matches with bindings {`H` $\mapsto$ `dog`, `T` $\mapsto$ `[dog,ape]`}

6. Evaluate `member(dog, [dog,ape])`

7. The first clause matches with bindings {`H` $\mapsto$ `dog`, `H1` $\mapsto$ `dog`}. The guard test in the first clause succeeds.

8. Evaluate `true` which is just `true`.

Note that each time a function clause is entered a fresh set of variable bindings is used, so that the values of the variable H and T in step 3 above are distinct from those in step 5.

3.3.7 Function bodies

Function bodies are sequences of expressions. The value of a sequence of expressions is obtained by sequentially evaluating each element in the sequence. The value of the body is the result of evaluating the last expression in the sequence.

For example, suppose we define a function to manipulate a bank account:

```
deposit(Who, Money) ->
    Old = lookup(Who),
    New = Old + Money,
    insert(Who, New),
    New.
```

The body of this function consists of a sequence of four statements. If we evaluate the expression `deposit(joe, 25)` then the function will be entered with bindings {`Who` $\mapsto$ `joe`, `Money` $\mapsto$ `10`}. Then `lookup(Who)` will be called. Assume this returns `W`. The return value (`W`) is matched against the free variable `Old`, the match succeeds. After this match we continue with the set of bindings {`Who` $\mapsto$ `joe`, `Money` $\mapsto$ `10`, `Old` $\mapsto$ `W`} ...

3.3.8 Tail recursion

A function call is tail-recursive if all the last calls in the function body are calls to other functions in the system.

For example, consider the following functions:

```
p() ->
    ...
    q(),
```

```
    ...

q() ->
    r(),
    s().
```

At some point in the execution of p the function q is called. The final function call in q is a call to s. When s returns it returns a value to q, but q does nothing with the value and just returns this value unmodified to p.

The call to s at the end of the function q is called a *tail-call* and on a traditional stack machine tail-calls can be compiled by merely jumping into the code for s. No return address has to be pushed onto the stack, since the return address on the stack at this point in the execution is correct and the function s will not return to q but to the correct place where q was called from in the body of p.

A function is tail-recursive if all possible execution paths in the function finish with tail-calls.

The important thing to note about tail-recursive functions is that they can run in loops without consuming stack space. Such function are often called "iterative functions."

Many functions can be written in either an iterative of non-iterative (recursive) style. To illustrate this, the factorial function can be written in these two different styles. Firstly the non tail-recursive way:

```
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
```

To write this in a tail-recursive manner requires the use of an additional function:

```
factorial(N) -> factorial_1(N, 1).

factorial_1(0, X) -> X;
factorial_1(N, X) -> factorial_1(N-1, N*X).
```

Many non-tail recursive functions can be made tail-recursive, by introducing an auxiliary function, with an additional argument.[6]

Many functions in Erlang are designed to run in *infinite loops*—in particular the client–server model assumes that the server will run in an infinite loop. Such loops must be written in a tail-recursive manner. A typical infinite loop in a server might be written something like:

```
loop(Dict) ->
    receive
       {store, Key, Value} ->
           loop(dict:store(Key, Value, Dict));
       {From, {get, Key}} ->
           From ! dict:fetch(Key, Dict),
           loop(Dict)
    end.
```

which is tail recursive.

3.3.9 Special forms

Two special forms are used for conditional evaluation of Erlang expression sequences. They are the `case` and `if` statements.

3.3.10 case

`case` has the following syntax:

```
case Expression of
  Pattern1 -> Expr_seq1;
  Pattern2 -> Expr_seq2;
  ...
end
```

[6] Called an accumulator.

`case` is evaluated as follows: Firstly, `Expression` is evaluated, assume this evaluates to *Value*. Thereafter *Value* is matched in turn against *Pattern1*, *Pattern2* ... etc. until a match is found. As soon as a match is found to some pattern *Pattern[i]* then the corresponding expression sequence *Expr_seq[i]* is evaluated—the result of evaluating the expression sequence *Expr_seq[i]* becomes the value of the case statement.

3.3.11 if

A second conditional primitive *if* is also provided. The syntax:

```
if
  Guard1 ->
    Expr_seq1;
  Guard2 ->
    Expr_seq2;
  ...
end
```

is evaluated as follows: Firstly *Guard1* is evaluated, if this evaluates to *true* then the value of *if* is the value obtained by evaluating the expression sequence *Expr_seq1*. If *Guard1* does not succeed *Guard2*... is evaluated until a match is found. At least one of the guards in the `if` statement must evaluate to `true` otherwise an exception will be raised. Often the final guard in an `if` statement is the atom `true` which guarantees that the last form in the statement will be evaluated if all other guards have failed.

3.3.12 Higher order functions

Higher order functions are functions which take functions as input arguments or produce functions as return values. An example of the former is the function `map` found in the `lists` module, which is defined as follows:

```
map(Fun, [H|T]) -> [Fun(H)|map(Fun, T)];
map(Fun, [])    -> [].
```

`map(F, L)` produces a new list by applying the function `F` to every element of the list `L`, so for example:

```
> lists:map(fun(I) -> 2 *I end, [1,2,3,4]).
[2,4,6,8]
```

Higher order functions can be used to create *control abstractions* for syntactic constructions that do not exist in the language.

For example, the programming language `C` provides a looping construct *for*, which can be used as follows:

```
sum = 0;
for(i = 0; i < max; i++){
  sum += f(i)
}
```

Erlang has no *for* loop but we can easily make one:

```
for(I, Max, F, Sum) when I < Max ->
    for(I+1, Max, F, Sum + F(I));
for(I, Max, F, Sum) ->
    Sum.
```

which could be used as follows:

```
Sum0 = 0,
Sum  = for(0, Max, F, Sum0).
```

Functions which *return* new functions can also be defined. The following example in the Erlang shell illustrates this:

```
1> Adder = fun(X) -> fun(Y) -> X + Y end end.
#Fun<erl_eval.5.123085357>
2> Adder10 = Adder(10).
#Fun<erl_eval.5.123085357>
3> Adder(10).
15
```

Here the variable `Adder` contains a function of `X`; evaluating `Adder(10)` binds `X` to `10` and returns the function `fun(Y) -> 10 + Y end.`

With devilish ingenuity recursive functions can also be defined, for example, factorial:

```
6> Fact = fun(X) ->
            G = fun(0,F)  -> 1;
                   (N, F) -> N*F(N-1,F)
                end,
            G(X, G)
          end.
#Fun<erl_eval.5.123085357>
7> Fact(4).
24
```

Functions can be referred to with the syntax `fun Name/Arity`. For example, the expression:

```
X = fun foo/2
```

is shorthand for writing:

```
X = fun(I, J) -> foo(I, J) end
```

where `I` and `J` are free variables which do not occur at any other place in the function where `X` is defined.

3.3.13 List comprehensions

List comprehensions are expression which generate lists of values. They are written with the following syntax:

```
[X || Qualifier1, Qualifier2, ...]
```

`X` is an arbitrary expression, and each qualifier is either a generator or a filter.

- Generators are written as `Pattern<-ListExpr` where `ListExpr` must be an expression which evaluates to a list of terms.
- Filters are either predicates or boolean expressions.

As an example, the well-known quicksort algorithm can be expressed in terms of two list comprehensions:

```
qsort([]) -> [];
qsort([Pivot|T]) ->
        qsort([X||X<-T,X =< Pivot]) ++
        [Pivot] ++
        qsort([X||X<-T,X > Pivot]).
```

Where ++ is the infix append operator.

If you are interested in crossword puzzles and need to compute all permutations of a string then you could use the function `perms` where:

```
perms([]) -> [[]];
perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

Where the infix operator `X--Y` is a copy of the list `X` where any element of `X` which also occurs in `Y` have been removed.

So for example:

```
> perms("123").
["123","132","213","231","312","321"]
```

3.3.14 Binaries

Binaries are memory buffers designed for storing untyped data. Binaries are used primarily to store large quantities of unstructured data and for efficient I/O operations. Binaries store data in a much more space-efficient manner than in lists or tuples. For example, a string stored as a list needs eight bytes of storage per character, whereas a string stored in a binary needs only one byte per character plus a small constant overhead.

The BIF `list_to_binary` converts a io-list to a binary, the inverse is computed with `binary_to_list`; `term_to_binary` converts an arbitrary term to a binary, the inverse is `binary_to_term`.

Note: An io-list is a list whose elements are either small integers (a small integer is an integer in the range 0 to 255) or binaries, or io-lists. The BIF `list-to_binary(A)` flattens the io-list and produces a binary constructed from this list. `binary_to_list/1` returns a flat list of small integers. `binary_to_list` is only a strict inverse of `list_to_binary(A)` in the case where `A` is a flat list of small integers.

Lists of binaries can be concatenated with `concatenate_binaries`, and a single binary can be split into two binary with `split_binary`.

We can illustrate a number of operations on binaries, in the shell:

```
1 > B1=list_to_binary([1,2,3]).
<<1,2,3>>
2>  B2=list_to_binary([4,5,[6,7],[],[8,[9]],245]).
<<4,5,6,7,8,9,245>>
3 > B3=concat_binary([B1,B2]).
<<1,2,3,4,5,6,7,8,9,245>>
4> split_binary(B3,6).
{<<1,2,3,4,5,6>>, <<7,8,9,245>>}
```

Expression 1 converts the list `[1,2,3]` to a binary `B1`. Here the notation `<<I1,I2,...>>` represents the binary made from the bytes `I1,I2 ...`

Expression 2 converts an io-list to a binary.

Expression 3 combines two binaries `B1` and `B2` into a single binary `B3`, and expression 4 splits `B4` into two binaries.

```
5> B = term_to_binary({hello,"joe"}).
<<131,104,2,100,0,5,104,101,108,108,111,107,
  0,3,106,111,101>>
6> binary_to_term(B).
{hello,"joe"}
```

The BIF `term_to_binary` converts its argument into a binary. The inverse function is `binary_to_term` which reconstructs the term from the binary. The binary produced by `term_to_binary` is stored in the so-called "external term format." Terms which have been converted to binaries by using `term_to_binary` can be stored in files, sent in messages over a network etc and the original term from which they were made can be reconstructed later. This is extremely useful for storing complex data structures in files or sending complex data structures to remote machines.

3.3.15 The bit syntax

The bit syntax provides a notation for constructing binaries and for pattern matching on the contents of binaries. To understand how binaries are constructed I will give a number of examples in the shell:

```
1> X=1,Y1=1,Y2=255,Y3=256,Z=1.
1
2> <<X,Y1,Z>>.
<<1,1,1>>
3> <<X,Y2,Z>>.
<<1,255,1>>
4> <<X,Y3,Z>>.
<<1,0,1>>
5> <<X,Y3:16,Z>>.
<<1,1,0,1>>
6> <<X,Y3:32,Z>>.
<<1,0,0,1,0,1>>
```

In line 1 a few variables `X`, `Y1`..`Y3` and `Z` and defined. Line 2 constructs a binary from `X`,`Y1` and `Z` the result is just `<<1,1,1>>`.

In Line 3 `Y2` is 255 and the value of `Y2` is copied unchanged to the second byte of the binary. When we try to create a binary from `Y3` (which is 256) the value is truncated, since 256 will not fit into a single byte. The quantifier `:16` added in line 5 corrects this problem.

If we do not quantify the size of an integer it is assumed to take up 8 bits. Line 6 shows the effect of a 32-bit quantifier.

Not only can we specify the size of an integer, but also the byte order, so for example:

```
7> <<256:32>>.
<<0,0,1,0>>
8> <<256:32/big>>.
<<0,0,1,0>>
9> <<256:32/little>>.
<<0,1,0,0>>
```

Line 7 says create a binary from the integer 32 and pack this integer into 32 bits. Line 8 tells the system to create a 32-bit integer with "big endian" byte order and line 9 says use little endian byte order.

Bit fields can also be packed:

```
10> <<1:1,2:7>>.
<<130>>
```

creates a single byte binary from a 1-bit field followed by a 7-bit field.

Unpacking binaries is the opposite to packing them, thus:

```
11> <<X:1,Y:7>> = <<130>>.
<<130>>
12> X.
1
13> Y.
2
```

is the inverse of line 10.

The pattern machine operation on binaries was originally designed for processing packet data. Pattern matching is performed over a sequence of zero or more "segments". Each segment is written with the syntax:

```
Value:Size/TypeSpecifierList
```

Here `TypeSpecifierList` is a hyphen-separated list of items of the form `End-Sign-Type-Unit`, where:

- `End` — specifies the endianess of the machine and is one of `big`, `little` or `native`.
- `Sign` — is one of `signed` or `unsigned`.
- `Type` — is one of `integer`, `float`, or `binary`.
- `Unit` — is of the form `unit:Int` where `Int` is a literal integer in the range `1..256`. The total size of the segment is `Size x Int` bits long, this size must be a multiple of eight bits.

Any of the above items may be omitted and the items can occur in any order.

A nice example of the use of binaries can be found in the Erlang Open Source Distribution [34] in the section entitled "Erlang Extensions Since 4.4." This shows how to parse an IP version four datagram in a single pattern-matching operation.

```
1  -define(IP_VERSION, 4).
2  -define(IP_MIN_HDR_LEN, 5).
3
4  ...
5  DgramSize = size(Dgram),
6  case Dgram of
7    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
8      ID:16, Flgs:3, FragOff:13,
9      TTL:8, Proto:8, HdrChkSum:16,
10     SrcIP:32,
11     DestIP:32, RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize ->
12         OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
13         <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
14         ...
```

Lines 7–11 match the IP datagram in a single pattern-matching expression. The pattern is complex spreading over three lines and illustrating how data which does not fall on byte boundaries can easily be extracted (for example, the `Flgs` and `FragOff` fields which are 3 and 13 bits long respectively).

Having pattern matched the IP datagram, the header and data part of the datagram can be isolated (lines 12–13).

3.3.16 Records

Records provide a method for associating a name with a particular element in a tuple. The problem with tuples is that when the number of elements in a tuple becomes large, it is difficult to remember which element in the tuple means what.

In a small tuple this is rarely a problem, so we often see programs which manipulated tuples with a small number of elements in the tuple.

As the number of elements in the tuples becomes greater it becomes more and more difficult to keep track of the meanings of the individual elements in the tuple. When the number of elements in the tuple is large, or if we wish to name the elements in the tuple for other purposes[7] then we can use records instead of tuples.

Record declarations are written with the syntax:

```
-record(Name, {
                Key1 = Default1,
                Key2 = Default2,
                ...
              }).
```

Here `Name` is the name of the record. `Key1`, `Key2` ... are the names of the fields in the record. Each field in a record can have a default value which is used if no value for this particular field is specified when the record is created.

For example, we might define a `person` record as follows:

```
-record(person, {
                 firstName="",
                 lastName = "",
                 age}).
```

Once a record has been defined, instance of the record can be created. For example:

[7] for example, documentation.

```
Person = #person{firstName="Rip",
                 lastname="Van Winkle",
                 age=793
                 }
```

creates a "Rip Van Winkel"[8] person.

We can write functions which pattern match on the fields of a record and which create new records, so when Mr. Van Winkel has his birthday we can call:

```
birthday(X=#person{age=N}) ->
   X#person{age=N+1}.
```

If the clause head matches the record then `X` is bound to the entire record, and `N` is bound to the `age` field of the record. `X#person{age=K}` makes a copy of `X` replacing the old value of the field named `age` in `X` with the value `K`.

3.3.17 epp

Before an Erlang module is compiled it is processed by the Erlang pre-processor `epp`. The Erlang pre-processor performs macro expansion and inserts any necessary include files.

The output of the pre-processes can be saved in a file by giving the command `compile:file(M, ['P'])`. This compiles any code in the file `M.erl` and produces a listing in the file `M.P` where all macros have been expanded and any necessary include files have been included.

3.3.18 Macros

Erlang macros are written:

```
-define(Constant, Replacement).
-define(Func(Var1, Var2,.., Var), Replacement).
```

[8] Even Mr. Google did not know how old Mr. Van Winkel was, so 793 is pure guesswork.

Macros are expanded by the Erlang pre-processor epp when an expression of the form ?MacroName is encountered. Variables occurring in macro definition match complete forms in the corresponding site of the macro call.

```
-define(macro1(X, Y), {a, X, Y}).

foo(A) ->
    ?macro1(A+10, b)
```

expands into:

```
foo(A) ->
    {a,A+10,b}.
```

The argument in a call to a macro and the return value from a macro must be a complete, well-balanced expression. Thus it is not possible to use macros like the following:

```
-define(start, {).
-define(stop,  }).

foo(A) ->
    ?start,a,?stop.
```

In addition, there are a number of predefined macros which provide information about the current module. They are as follows:

- ?FILE expands to the current file name.
- ?MODULE expands to the current module name.
- ?LINE expands to the current line number.

3.3.19 Include files

Files can be included with the syntax:

```
-include(Filename).
```

Conventional include files have the extension `.hrl`. The `FileName` should contain an absolute or relative path so that the preprocessor can locate the appropriate file.

Library header files can be included with the syntax

```
-include_lib(Name).
```

For example:

```
-include_lib("kernel/include/file.hrl").
```

In which case the Erlang compiler will find the appropriate include files.

3.4 Concurrent programming

In Erlang, creation of a parallel process is achieved by evaluating the `spawn` primitive. The expression:

```
Pid = spawn(F)
```

Where `F` is a fun of arity zero creates a parallel process which evaluates `F`. Spawn returns a process identifier (Pid) which can be used to access the newly created process.

The syntax `Pid ! Msg` sends the message `Msg` to `Pid`. The message can be received using the `receive` primitive, with the following syntax:

```
receive
  Msg1 [when  Guard1] ->
    Expr_seq1;
  Msg2 [when  Guard2] ->
    Expr_seq2;
  ...
  MsgN [when GuardN] ->
    Expr_seqN;
  ...
  [; after TimeOutTime ->
    Timeout_Expr_seq]
end
```

`Msg1...MsgN` are patterns. The patterns may be followed by optional guards. When a message arrives at a process it is put into a mailbox belonging to that process. The next time the process evaluates a `receive` statement the system will look in the mailbox and try to match the first item in the mailbox with the set of patterns contained in the current `receive` statement. If no message matches then the received message is moved to a temporary "save" queue and the process suspends and waits for the next message. If the message matches and if any guard test following the matching pattern also matches then the sequence of statements following the match are evaluated. At the same time, any saved messages are put back into the input mailbox of the process.

The `receive` statement can have an optional timeout. If no matching message is received within the timeout period then the commands associated with the timeout are evaluated.

3.4.1 register

When we send a message to a process, we need to know the name of the process. This is very secure, but somewhat inconvenient since all processes which want to send a message to a given process have to somehow obtain the name of that process.

The expression:

```
register(Name, Pid)
```

creates a global process, and associates the atom `Name` with the process identifier `Pid`. Thereafter, messages sent by evaluating `Name!Msg` are sent to the process `Pid`.

3.5 Error handling

Evaluating a function in Erlang will result in exactly one of two possible outcomes: either the function will return a value, or, it will generate an exception.

Generating exceptions is either implicit (that is, generated by the Erlang run-time system) or is explicitly generated by evaluating the primitive `exit(X)`. Implicit exceptions are described in the next section.

Here is an example of implicit exception generation. Suppose we write:

```
factorial(0) -> 1;
factorial(N) -> N*factorial(N-1).
```

Evaluating `factorial(10)` returns the *Value* `3628800`, but evaluating `factorial(abc)` raises the exception `{'EXIT',{badarith,...}}`. Exceptions cause the program to stop what it is doing and do something else—that is why they are called exceptions. If we write:

```
J = factorial(I)
```

we expect the value of `J` to be the value of `factorial(I)` if `I` is an integer. If `factorial` is called with a non-integer argument the statement makes no sense. The program fragment:

```
I = "monday",
J = factorial(I),
```

is nonsense, since we cannot compute `factorial("monday")`. `J` thus has no value and it is pointless to proceed.

Many programming languages ignore the distinction between values and exceptions and blithely continue even though the program is nonsense.

3.5.1 Exceptions

Exceptions are abnormal conditions which are detected by the Erlang run-time system. Erlang programs are compiled to virtual machine instructions which are executed by a virtual machine emulator which is part of the Erlang run-time system.

If the emulator detects a condition where it cannot decide what to do, it generates an exception. There are six types of exception:

1. Value errors — these are things like "divide by zero". Here an argument to a function has the correct type, but an incorrect value.

2. Type errors — these are generated when an Erlang BIF is called with an argument which has an incorrect type. For example, the BIF `atom_to_list(A)` converts the atom `A` to a list of the ASCII integer codes which represent the atom. If `A` is not an atom, the run-time system generates an exception.

3. Pattern-matching errors — these are generated when an attempt is made to match a data structure against a number of patterns and no matching pattern is found. This can occur in function head matching, or in matching the alternatives in a `case`, `receive` or `if` statement.

4. Explicit exits — these are generated explicitly by evaluating the expression `exit(Why)`, which generates the exception `Why`.

5. Error propagation — if a process receives an exit signal it may decide to die and propagate the exit signal to all processes in its link set (see section 3.5.6).

6. System exceptions — the run-time system might terminate a process if it runs out of memory or if it detects an inconsistency in some internal table. Such exceptions are outside the programmer's control.

3.5.2 catch

Exceptions can be converted to values by using the `catch` primitive. We can illustrate this in the Erlang shell by trying to evaluate an illegal expression whose evaluation leads to the generation of an exception. We will try to bind the value of `1/0` to the free variable `X`. This is what happens:

```
1> X = 1/0.

=ERROR REPORT==== 23-Apr-2003::15:20:43 ===
Error in process <0.23.0> with exit value:
{badarith,[{erl_eval,eval_op,3},{erl_eval,expr,3},
{erl_eval,exprs,4},{shell,eval_loop,2}]}
** exited: {badarith,[{erl_eval,eval_op,3},
                      {erl_eval,expr,3},
                      {erl_eval,exprs,4},
                      {shell,eval_loop,2}]} **
```

Here entering the expression `X = 1/0` in the Erlang shell caused an exception to be generated and an error message is printed to standard output. If we try to print the value of the variable `X` we see the following:

```
2> X.
** exited: {{unbound,'X'},[{erl_eval,expr,3}]} **
```

`X` has, of course, no value, so another exception is generated and another error message printed.

To convert the exception into a value we evaluate it within a `catch` statement, as follows:

```
3> Y = (catch 1/0).
{'EXIT',{badarith,[{erl_eval,eval_op,3},
                   {erl_eval,expr,3},
                   {erl_eval,exprs,4},
                   {shell,eval_loop,2}]}}
```

Now Y has a value, namely a 2-element tuple whose first element is the atom EXIT, and whose second value is the term {badarith,...}. Y is a normal Erlang term, which can be freely examined and manipulated like any other Erlang data structure. The expression:

```
Val = (catch Expr)
```

evaluates Expr in some context. If the function evaluation terminates normally, then catch returns the value of this expression. If an exception occurs during the evaluation then evaluation of the function stops immediately and an exception is generated. The exception is an Erlang object which describes the problem, in this case the value of catch is the value of the exception that was generated.

If evaluating (catch Expr) returns a term of the form {'EXIT',W} then we say that the expression terminated with reason W.

If an exception is generated anywhere inside a catch then the value of catch is just the value of the exception. If an exception is generated outside the scope of a catch then the process in which the exception was generated will die and the exception will be broadcast to any processes which are currently linked to the dying process. Links are created by evaluating the BIF link(Pid).

3.5.3 exit

Explicit exceptions can be generated by calling the exit primitive. Here is an example:

```
sqrt(X) when X < 0 ->
    exit({sqrt,X});
sqrt(X) ->
    ...
```

which generates the exception {sqrt,X} if called with a negative argument X.

3.5.4 throw

The primitive `throw` is used to change the syntactic form of an exception.

- If an exception is generated by calling `exit(P)` within the scope of some function `F` then the result of evaluating `(catch F)` will be a term of the form `{'EXIT',P}`.
- If an exception is generated by calling `throw(Q)` within the scope of some function `F` then the result of evaluating `(catch F)` will be the term `Q`.

Throw can be used for distinguishing user-generated exceptions from exceptions produced by the run-time system.

3.5.5 Corrected and uncorrected errors

Suppose we write:

```
g(X) ->
    case (catch h(X)) of
        {'EXIT', _} ->
            10;
        Val ->
            Val
    end.

h(cat) -> exit(dog);
h(N)   -> 10*N.
```

Evaluating `h(cat)` generates an exception, `h(20)` returns the value `200`. Evaluating `g(cat)` or `g(dog)` returns the value `10` whereas `g(10)` returns the value `100`.

When we evaluate `g(cat)` the following sequence of events occurs:

1. `h(cat)` is evaluated.

2. h generates an exception.

3. The exception is caught in g.

4. g returns a value.

Evaluating g(dog) causes the following to occur:

1. h(dog) is evaluated.

2. N is bound to dog in the body of h.

3. N*10 is evaluated where N = dog.

4. An exception is generated in '*'.

5. The exception is propagated to h.

6. The exception is caught in g.

7. g returns a value.

If we look at this carefully we observe that in evaluating d(dog) an exception was raised but was caught and corrected in g.

Here we can say that an error did occur but that it was corrected.

If we had evaluated h(dog) directly then an error would have occurred that was not caught and was not corrected.

3.5.6 Process links and monitors

When one process in the system dies we would like other processes in the system to be informed; recall (page 27) that we need this in order to be able to program fault-tolerant systems. There are two ways of doing this. We can use either a process link or a process monitor.

Process links are used to group together sets of processes in such a way that if an error occurs in any one of the processes then all the processes in the group get killed.

Process monitors allow individual processes to monitor other processes in the system.

Process links

The primitive `catch` is used to contain errors which occur *within* a process. We now ask what happens if the top-level `catch` in a program does not manage to correct an error which it has detected? The answer is that the processes itself terminates.

The reason for failure is just the argument of the exception. When a process fails the reason for failure is *broadcast* to all processes which belong in the so-called "link set" of the failing process. A process `A` can add the process `B` to its link set by evaluating the BIF `link(B)`. Links are symmetric, in the sense that if `A` is linked to `B` then `B` will also be linked to `A`.

Links can also be created when a process is created. If the process `A` creates the process `B` by evaluating:

```
B = spawn_link(fun() -> ... end),
```

then the process `B` will be linked to `A`. This is semantically equivalent to evaluating `spawn` immediately followed by `link`, only the two expression are evaluated atomically, instead of sequentially. The `spawn_link` primitive was introduced to correct a rare programming error which can occur if a process dies immediately during the spawning process and does not reach the `link` statement.[9]

If a process `P` dies with an uncaught exception `{'EXIT',Why}` then the exit signal `{'EXIT',P,Why}` will be sent to all processes in the link set of the process `P`.

So far I have not mentioned signals. Signals are things which are sent between processes when a process terminates. The signal is a tuple of the form `{'EXIT',P,Why}` where `P` is the Pid of the process which has terminated and `Why` is a term describing the reason for termination.

Any process which receives an exit signal will die if `Why` is not the atom `normal`. There is one exception to this rule: if the receiving process is a system process, then it will not die but instead the signal will be converted

[9]This could happen for example, if a process tries to spawn a processes which uses code in a module which does not yet exist.

into a normal inter-process message and be added to the input mailbox of the process. By evaluating the BIF `process_flag(trap_exit,true)` a normal process can become a system process.

A typical code fragment for a system process which can handle failures in other processes is something like this:

```
start() -> spawn(fun go/0).

go() ->
    process_flag(trap_exit, true),
    loop().

loop() ->
    receive
        {'EXIT',P,Why} ->
            ... handle the error ...
    end
```

One additional primitive completes the picture. `exit(Pid,Why)` sends an exit signal to the process `Pid` with reason `Why`. The process evaluating `exit/2` does not terminate, so such a message can be used to "fake" the death of a process.[10]

Again there is one exception to the rule that a system process will convert all signals into messages; evaluating `exit(P,kill)` sends an *unstoppable exit* to the process `P` which will be terminated with extreme prejudice. This use of `exit/2` is needed to kill processes which refuse to honour requests to voluntarily terminate.

Process links are useful for setting up groups of processes, which will all die if anything goes wrong in any of the processes. Usually we just link together all the processes which belong to an application, and let one of the processes assume a "supervisor" role. The supervisor process is set to trap exits. If anything goes wrong then the entire group will die except the supervisor process which can receive messages which inform it about the failures of the other processes in the group.

[10]This is a feature, not a bug!

Process Monitors

Process links are useful for entire groups of processes, but not for monitoring pairs of processes in an asymmetric sense. In a typical client-server model, the relationship between the client and the servers is asymmetric as regards error handling. Suppose that a server is involved in a number of long-lived sessions with a number of different clients; if the server crashes, then we want to kill all the clients, but if an individual client crashes we do not wish to kill the server.

The primitive `erlang:monitor/2` can be used to set up a monitor. If process `A` evaluates:

```
Ref = erlang:monitor(process, B)
```

Then if `B` dies with exit reason `Why`, `A` will be sent a message of the form:

```
{'DOWN', Ref, process, B, Why}
```

Neither `A` nor `B` have to be system processes in order to set up or receive monitor messages.

3.6 Distributed programming

Erlang programs can be easily ported from a uni-processor to a collection of processors. Each complete and self-contained Erlang system is called a *node*. One or more Erlang nodes can run within a host operating system. Testing a distributed application is simplified by the fact that many Erlang nodes can be run on the same operating system. A distributed application can be developed and tested by running all the nodes in the application on a single processor. When the application works the different nodes which were assigned to the same processor can be moved to different nodes in a network of distributed processors. With the exception of timing all operations in the distributed system should work in exactly the same way as they worked in the single-node test system.

Two primitives are needed for distributed processing:

- `spawn(Fun,Node)` — spawns a function `Fun` on the remote node `Node`.
- `monitor(Node)` — is used for monitoring the behaviour of an entire node.

  `monitor` is analogous to `link`, the difference being that the controlled object is an entire node instead of a process.

3.7 Ports

Ports provide a mechanism for Erlang programs to communicate with the outside world. Ports are created by evaluating the BIF `open_port/2`. Associated with each port is a so-called "controlling-process". The controlling processes is said to *own* the port. All messages from the port are sent to the controlling process and only the controlling process is allowed to send messages to the port. The controlling process is initially the process which created the port, but this process can be changed.

If `P` is a port, and `Con` is the Pid of the port's controlling process then the port can be commanded to do something by evaluating the expression:

```
P ! {Con, Command}
```

where `Command` can have one of the following three possible values:

- `{command,Data}` — sends `Data` to the external object. `Data` must be an io-list (see page 59 for a definition of io-lists). The io-list is flattened and the data elements in the list are sent to the external application.
- `close` — closes the port. The port must respond by sending a `{P,closed}` message to the controlling process.
- `{connect,Pid1}` — changes the controlling process to `Pid`. The port must send a `{Port,connected}` to the original controlling process, thereafter all new messages are sent to the new controlling process.

All data from the external application results in `{Port,{data,D}}` messages being sent to the controlling process for the port.

The exact format of the messages and how the messages are framed depends upon the details of how the port was opened. See [34] for more details.

3.8 Dynamic code change

Erlang supports a simple mechanism for dynamic code changes. In a running Erlang node all processes share the same code. We have to consider, therefore, what happens if we change the code in a running system.

In a sequential programming language there is only one thread of control, so if we wish to dynamically change the code we only have to worry about what is happening in that single thread of control. Usually in a sequential system, if we wish to change the code, we stop the system, change the code and re-start the program.

In a real-time control system, however, we often do not wish to stop the system in order to change the code. In certain real-time control systems, we might never be able to turn off the system in order to change the code and so these systems have to be designed so that the code can be changed without stopping the system. An example of such a system is the X2000 satellite control system [2] developed by NASA.

The Erlang system allows for two versions of code for every module. If the code for a particular module has been loaded then all new processes that call any of this code will be dynamically linked with the latest version of the module. If the code for a particular module is subsequently changed then processes which execute code in this module can choose either to continue executing the old code for the module, or to use the new code. The choice is determined by how the code is called.

If the code is called with a fully-qualified name, that is a module name followed by a function name, then the latest version of the module will always be called, otherwise the current version will be called. For example, suppose we write a server loop as follows:

```
-module(m).
```

```
...
loop(Data, F) ->
  receive
    {From, Q} ->
        {Reply, Data1} = F(Q, Data),
        m:loop(data1, F)
  end.
```

The first time the module `m` is called the code for the module will be loaded. At the end of the loop `m:loop` is called. Since there is only one version of the module the code in the current module will be called.

Suppose now that we change the code for the module `m` and we re-compile and re-load the code for `m`. If we do this, when `m:loop` is called in the last clause of the `receive` statement, the code in the new version of `m` will be called. Note that it is the programmer's responsibility to ensure that the new code to be called is compatible with the old code. It is also highly advisable that all code replacement calls are tail-calls (see section 3.3.8), since in a tail-call there is no old code to return to, and thus after a tail-call all old code for a module can safely be deleted.

If we want to carry on executing code in the current module and not change to code in the new module, then we would write the loop without the fully qualified call, thus:

```
-module(m).
...
loop(Data, F) ->
  receive
    {From, Q} ->
        {Reply, Data1} = F(Q, Data),
        loop(data1, F)
  end.
```

In this case code in the new version of the module will not be called.

Judicious use of the mechanism allows processes to execute both old and new versions of code in different modules at the same time.

Note there is a limit to two versions of the code. If a third attempt is made to re-load the module then all processes executing code in the first module will be killed.

In addition to these calling conventions there are a number of BIFs that have to do with code replacement. These are fully described in [5].

3.9 A type notation

If we build a software module, how can we describe how it is to be used? The conventional answer involves the use of a so-called programming API (Application Programming Interface). The API is usually a list of those functions in the module which are externally callable and the types of the inputs to the function and the types of the output.

Here is an example of how the types of a number of functions can be specified in the Erlang type notation:

```
+type file:open(fileName(), read | write) ->
        {ok, fileHandle()}
      | {error, string()}.

+type file:read_line(fileHandle()) ->
        {ok, string()}
      | eof.

+type file:close(fileHandle()) ->
        true.

+deftype fileName()   = [int()]
+deftype string()     = [int()].
+deftype fileHandle() = pid().
```

Each of the Erlang primitive data types has its own type. These primitive types are:

- `int()` — is the integer type.

- `atom()` — is the atom type.
- `pid()` — is the Pid type.
- `ref()` — is the reference type.
- `float()` — is the Erlang float type.
- `port()` — is the port type.
- `bin()` — is the binary type.

List, tuple and alternation types are defined recursively:

- `{T1,T2,...,Tn}` is the *tuple type* if `T1 .. Tn` are types. We say that `{X1,X2,...,Xn}` is of type `{T1,T2,...,Tn}` if `X1` is of type `T1` and `X2` is of type `T2`, ... and `Xn` is of type `Tn`.
- `[T]` is the *list type* if `T` is a type. We say that `[X1,X2,...,Xn]` is of type `[T]` if all the `X`'s are of type `T`. Note that the empty list `[]` is of type `[T]` for all values of `T`.
- `T1|T2` is the *alternation type* if `T1` and `T2` are types. We say that `X` is of type `T1|T2` if `X` is of type `T1` or if `X` is of type `T2`.

New types are introduced with the notation:

```
+deftype name1() = name2() = ... = Type.
```

The names `name1`, `name2`,... follow the syntax of Erlang atoms.
Type variables are written using the syntax of Erlang variables.
so, for example, we can define:

```
+deftype bool()    = true | false.
+deftype weekday() = monday|tuesday|wednesday|
                     thursday|friday.
+deftype weekend() = saturday() | sunday().
+deftype day()     = weekday() | weekend().
```

Function types are written:

```
+type functionName(T1, T2, ..., Tn) -> T.
```

Where all the T's are types. If a type variable occurs more than once in a type definition then all the instances of the type at the position implied by the variable must have the same type.

Here are some examples:

```
+deftype string() = [int()].
+deftype day()    = number() = int().
+deftype town()   = street() = string().

+type factorial(int())  -> int().
+type day2int(day())    -> int().
+type address(person()) -> {town(), street(), number()}.
```

Finally, function types are written:

```
+type fun(T1, T2, ..., Tn) -> T end
```

so, for example, the type of `map/2` can be written:

```
+type map(fun(X) -> Y end, [X]) -> [Y].
```

The type notation here is a much simplified version of the notation developed by Wadler & Marlow [49].

3.10 Discussion

This chapter has introduced a significant subset of Erlang—sufficient, at least, to understand all the examples in the thesis, but I have not yet addressed the question *"Is Erlang a suitable language for programming fault-tolerant systems in?"* I believe the answer is *"Yes."* I argued earlier

that in order to program a fault-tolerant system the programming language used for the job had to satisfy certain properties (R1–R6 on page 27). I now claim that Erlang does indeed satisfy these properties, for the following reasons:

- Process are fundamental to Erlang so R1 is satisfied.
- R2 is satisfied since processes in Erlang are designed as units of error encapsulation. If one process terminates because of a software error, other process executing in the same Erlang node will not be affected (unless, of course, they have been linked to the terminating process, in which case the interaction is intentional).
- Processes fail immediately if functions within the processes are called with incorrect arguments, or if the system BIFs are called with incorrect arguments. Immediate failure corresponds to Gray's notion of *fail-fast processes* (page 34), to Schneider's notion of a *fail-stop processor* (page 34) and to Renzel's statement that we must detect errors and fail as early as possible (page 35).
- When processes fail, the reason for failure is broadcast to the current link set of the process, thus satisfying R3 and R4.
- R5 is satisfied by the code upgrade mechanism described in section 3.8.
- R6 is not satisfied in Erlang, but it is satisfied in the Erlang libraries. Stable storage makes use of either `dets` or `mnesia`. `dets` is a single system disk-based storage system. If a process or node crashes then data stored in `dets` should survive the crash. For stronger data protection the data should be stored on two physically separated nodes, using the `mnesia` data base, which is one of the OTP applications.

I also note in passing that Schneider's "halt on failure," "Failure status property" and "Stable storage property" (page 34) are satisfied either directly in Erlang itself, or in the Erlang libraries.

4 Programming Techniques

The previous chapter was about Erlang, but not about how to program in Erlang. This chapter is about Erlang programming techniques. The programming techniques are concerned with:

- *Abstracting out concurrency* — concurrent programs are, in some sense, more difficult to write than sequential programs. Instead of writing one module which has both concurrent and sequential code I show how to structure the code into two modules, one of which has all the concurrent code, the other having only purely sequential code.

- *Maintaining the Erlang view of the world* — in Erlang terms *everything is a process.* To help maintain this view I introduce the idea of a protocol converter, which helps the programmer maintain the illusion that everything is an Erlang process.

- *The Erlang view of errors* — the treatment of how errors are handled in Erlang is radically different from most other programming languages, I show how error situations should be programmed in Erlang.

- *Intentional programming* — this is a programming style where the programmer can easily see from the code exactly what the programmer intended, rather than by guessing at the meaning from a superficial analysis of the code.

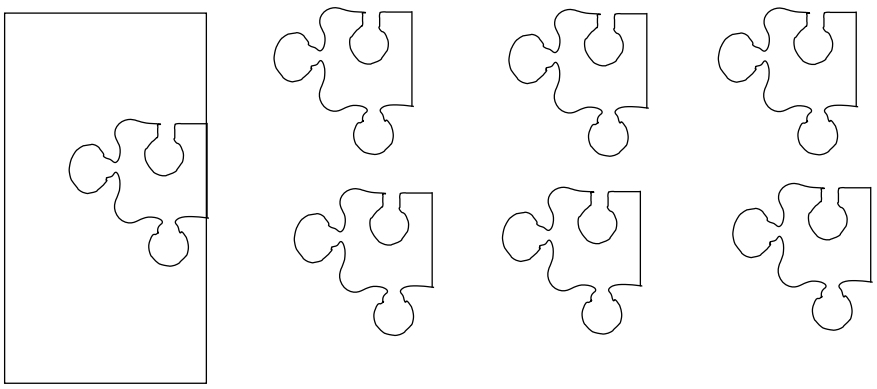


Figure 4.1: A generic component with plugins. All the concurrency and fault-handling are handled in the generic component. The plugins are written with purely sequential code.

4.1 Abstracting out concurrency

When we program we want to structure the code into "difficult" and "easy" modules. The difficult modules should be few and written by expert programmers. The easy modules should be many and written by less experienced programmers. Figure 4.1 shows a generic component (the difficult part), and a number of "plugins" (the easy parts) which are used to parameterise the generic component.

The generic component should hide details of concurrency and mechanisms for fault-tolerance from the plugins. The plugins should be written using only sequential code with well-defined types.

In the following I show how to structure a client-server into a generic component and a number of plugins.

Structuring a system into generic component and plugins is a common programming technique—what is unusual in our approach is that the generic component can provide a rich environment in which to execute the plugin. The plugin code itself can have errors, and the code in the plug-in can be dynamically modified, the entire component can be moved in a network, and all this occurs without any explicit programming in the plugin code.

Abstracting out concurrency is one of the most powerful means available for structuring large software systems. Despite the ease with which concurrent programs can be written in Erlang it is still desirable to restrict code which explicitly handles concurrency to as few modules as possible.

The reason for this is that concurrent code cannot be written in a side-effect free manner, and as such, is more difficult to understand and analyse than purely sequential side-effect free code. In a system involving large numbers of processes, issues of message passing ordering and potential dead- or live-lock problems can make concurrent systems very difficult to understand and program.

The most common abstraction used by applications written in Erlang is the client–server abstraction. In virtually all systems that are programmed in Erlang, use of the client-server abstraction far outweighs the use of any other abstraction. For example, on page 174 we see that 63% of all the behaviours used in the AXD301 system were instances of the `gen_server` behaviour which provides a client-server abstraction.

I will start with a simple universal client-server `server1` and then show how it can be parameterised to form a simple name service.

I will also extend the simple server in two ways. Firstly I modify the basic server to form a *fault tolerant* server `server2`, then I extend it to a version providing *dynamic code upgrade* (`server3`). The step-wise refinement of the server code, `server1 -> server2 -> server3` leads eventually to `gen_server` which is one of the standard behaviours in the OTP libraries. The code for `gen_server` performs many more operations than are performed in the simple server examples shown here. The principles, however, of the `gen_server` are the same as for the simple server examples; namely that the client-server is separated into a generic part, which takes care of the concurrency, and a sequent plug-in modules which merely parameterises the generic server in a particular manner to create a specific instance of a server.

The treatment of both these extensions is deliberately simplistic. I have ignored many implementation issues in favour of a simple explanation which demonstrates the principles involved.

The client-server model is illustrated in figure 4.2. The client-server model is characterized by a central server and an arbitrary number of

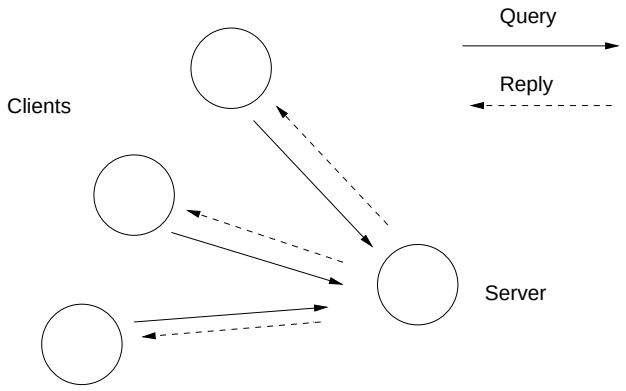


Figure 4.2: Client server

clients. The client server model is generally used for resource management operations. We assume that several different clients want to share a common resource and that the server is responsible for managing the resource.

If we ignore how the server is started and stopped and ignore all error cases then it is possible to describe the server by means of a single function `F`.

Let us suppose that the server is in a state `State` and that it receives a message `Query` from some client. The server should respond to this query by returning a message `Reply` to the client and changing state to `State1`.

These values are completely determined by the server function `F` and are computed by the Erlang expression:

```
{State1, Reply} = F(Query, State)
```

which is evaluated within the server.

The first universal server `server1.erl` is shown in figure 4.3. The client stub routine `rpc` (lines 13–17) sends a messages to the server (line 14) and waits for a reply (lines 15–17). The server receives the message sent by the client stub (line 23), computes a reply and a new state (line 24), sends the reply back to the server (line 25), and recursively calls itself (line 26). Note that the recursive call to `loop/3` (line 26) is a tail-call (see section 3.3.8), because it is a tail-call the variable `State` will no

```
-module(server1).

-export([start/3, stop/1, rpc/2]).

start(Name, F, State) ->
    register(Name,
             spawn(fun() ->
                           loop(Name, F, State)
                   end)).

stop(Name) ->  Name ! stop.

rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, Reply} -> Reply
    end.

loop(Name, F, State) ->
    receive
        stop ->
            void;
        {Pid, Query} ->
            {Reply, State1} = F(Query, State),
            Pid ! {Name, Reply},
            loop(Name, F, State1)
    end.
```

Figure 4.3: A simple server.

longer be accessible by any code and thus any storage accessed by `State` that cannot be reached from `State1` will eventually be reclaimed by the garbage collector. `loop/3` is said therefore to run in constant space, apart, that is, from the local storage (stored in the variable `State`) which is needed to store the state of the server itself. `server1.erl` exports three routines:

- `start(Name, Fun, State)` — starts a server with name `Name`. The initial state of the server is `State`, `Fun` is a function which completely characterises the behaviour of the server.
- `stop(Name)` — stops the server `Name`.
- `rpc(Name, Q)` — perform a remote procedure call on the server called `Name`.

We can use this server to implement a very simple "Home Location Register[1]" which we call VSHLR (Very Simple HLR). Our VSHLR has the following interface:

- `start()` — starts the HLR.
- `stop()` — stops the HLR.
- `i_am_at(Person,Loc)` — tells the HLR that `Person` is at the location `Loc`.
- `find(Person) -> {ok, Loc} | error` — tries to find the position of `Person` in the HLR. The HLR responds with `{ok, Loc}` where `Loc` is the last reported location, or, `error` if it doesn't know where the person is.

`vshlr1` can be implemented by parameterising `server1`, this is shown in figure 4.4.

Here is a simple session using the server:

[1]Home location registers are widely used in the telecoms industry to keep track of the current location of a mobile user.

```
-module(vshlr1).

-export([start/0, stop/0, handle_event/2,
         i_am_at/2, find/1]).

-import(server1, [start/3, stop/1, rpc/2]).
-import(dict,    [new/0, store/3, find/2]).

start() -> start(vshlr, fun handle_event/2, new()).

stop() -> stop(vshlr).

i_am_at(Who, Where) ->
    rpc(vshlr, {i_am_at, Who, Where}).

find(Who) ->
    rpc(vshlr, {find, Who}).

handle_event({i_am_at, Who, Where}, Dict) ->
    {ok, store(Who, Where, Dict)};
handle_event({find, Who}, Dict) ->
    {find(Who, Dict), Dict}.
```

Figure 4.4: A Very Simple Home Location Register.

```
1> vshlr1:start().
true
2> vshlr1:find("joe").
error
3> vshlr1:i_am_at("joe", "sics").
ack
4> vshlr1:find("joe").
{ok,"sics"}
```

Even though our VSHLR program is extremely simple, it illustrates and provides simple solutions to a number of design problems. The reader should note the following:

- There is a *total* separation of functionality into two different modules. *All* the code that has to do with spawning processes, sending and receiving messages etc is contained in `server1.erl`. All the code that has to do with the *implementation* of the VSHLR is contained in `vshlr1.erl`.

- The code in `vshlr1.erl` makes no use of any of the Erlang concurrency primitives. The programmer who writes this module needs to know nothing about concurrency or fault-handling.

The second point is very important. This is an example of *factoring out concurrency*—since writing concurrent programs is generally perceived as being difficult and since most programmers are more experienced in writing sequential code then being able to factor out the concurrency is a distinct advantage.

Not only can we factor out concurrency but we can mask possible errors in the code which is used to parameterise the server function. This is shown in the next section.

4.1.1 A fault-tolerant client-server

I now extend the server by adding code for error recovery, as shown in figure 4.5. The original server will crash if there is an error in the function

F/2. The term "fault-tolerance" usually applies to hardware, but here we mean the sever will tolerate faults in the function F/2 which is used to parameterise the server.

The function F/2 is evaluated within a catch and the client is killed if a RPC request is made which would have killed the server.

Comparing the new server with the old we note two small changes: the rpc code has changed to:

```
rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Reply} -> Reply
    end.
```

and the code in the inner section of the receive statement in loop/3:

```
case (catch F(Query, State)) of
    {'EXIT', Why} ->
        log_error(Name, Query, Why),
        From ! {Name, crash},
        loop(Name, F, State);
    {Reply, State1} ->
        From ! {Name, ok, Reply},
        loop(Name, F, State1)
end
```

Looking at these changes in more detail we observe that if evaluating the function in the server loop raises an exception then three things happen:

1. The exception is logged—in this case we just print out the exception but in a more sophisticated system we would log it on non-volatile storage.

```
-module(server2).
-export([start/3, stop/1, rpc/2]).

start(Name, F, State) ->
    register(Name,  spawn(fun() -> loop(Name,F,State) end)).

stop(Name) -> Name ! stop.

rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Reply} -> Reply
    end.

loop(Name, F, State) ->
    receive
        stop -> void;
        {From, Query} ->
            case (catch F(Query, State)) of
                {'EXIT', Why} ->
                    log_error(Name, Query, Why),
                    From ! {Name, crash},
                    loop(Name, F, State);
                {Reply, State1} ->
                    From ! {Name, ok, Reply},
                    loop(Name, F, State1)
            end
    end.

log_error(Name, Query, Why) ->
    io:format("Server ~p query ~p caused exception ~p~n",
              [Name, Query, Why]).
```

Figure 4.5: A simple server with error recovery.

2. A `crash` message is sent to the client. When the crash message is received by the client it will raise an exception in the client code. This is desirable since it is probably pointless for the client to continue.

3. The server continues to operate with the *old* value of the state variable. Thus the RPC obeys "transaction semantics" that is to say, it either succeeds in its entirety and the state of the server is updated, or, it fails, in which case the state of the server is unchanged.

Note that the code in `server2.erl` is only written to protect against an error in the characteristic function which parameterises the server. If the server itself dies (it could, for example, be deliberately killed by some other process in the system), then the client RPC stub will hang indefinitely, waiting for a reply message that will never be sent. If we wish to guard against this possibility, then we could write the RPC routine as follows:

```
rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Reply} -> Reply
    after 10000 ->
        exit(timeout)
    end.
```

This solution, while it solves one problem, introduces another problem: *What value should we choose for the timeout?*. A better solution, which I will not elaborate here, makes use of applications and supervision trees. Server failures, should be detected not by client software, but by special supervisor processes which are responsible for correcting the errors introduced by the failure of a server.

We can now run this server parameterised with a version of VSHLR (`vshlr2`) which contains a deliberate error:

A session using this modified server is shown below:

```
-module(vshlr2).

-export([start/0, stop/0, i_am_at/2, find/1]).

-import(server2, [start/3, stop/1, rpc/2]).
-import(dict,    [new/0, store/3, find/2]).

start() -> start(vshlr, fun handle_event/2, new()).

stop() -> stop(vshlr).

i_am_at(Who, Where) ->
    rpc(vshlr, {i_am_at, Who, Where}).

find(Who) ->
    rpc(vshlr, {find, Who}).

handle_event({i_am_at, Who, Where}, Dict) ->
    {ok, store(Who, Where, Dict)};
handle_event({find, "robert"}, Dict) ->
    1/0;
handle_event({find, Who}, Dict) ->
    {find(Who, Dict), Dict}.
```

Figure 4.6: A home location register with a deliberate error.

```
> vshlr2:start().
true
2> vshlr2:find("joe").
error
3> vshlr2:i_am_at("joe", "sics").
ok
4> vshlr2:find("joe").
{ok,"sics"}
5> vshlr2:find("robert").
Server vshlr query {find,"robert"}
caused exception {badarith,[{vshlr2,handle_event,2}]}
** exited: rpc **
6> vshlr2:find("joe").
{ok,"sics"}
```

Hopefully the information in the exception is sufficient to debug the program (satisfying requirement R3 on page 27).

As a final improvement we modify the server in figure 4.5 to allow us to change the code in the server "on-the-fly," this is shown in figure 4.7.

I will parameterise this with `vshlr3`. `vshlr3` is not shown here but it is identical to `vshlr2` with one exception; `server2` in line three of the module is replaced by `server3`.

```
1> vshlr3:start().
true
2> vshlr3:i_am_at("joe", "sics").
ok
3> vshlr3:i_am_at("robert", "FMV").
ok
4> vshlr3:find("robert").
Server vshlr query {find,"robert"}
caused exception {badarith,[{vshlr3,handle_event,2}]}
** exited: rpc **
5>  vshlr3:find("joe").
{ok,"sics"}
```

```
-module(server3).
-export([start/3, stop/1, rpc/2, swap_code/2]).

start(Name, F, State) ->
    register(Name, spawn(fun() -> loop(Name,F,State) end)).

stop(Name) -> Name ! stop.

swap_code(Name, F) -> rpc(Name, {swap_code, F}).

rpc(Name, Query) ->
    Name ! {self(), Query},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Reply} -> Reply
    end.

loop(Name, F, State) ->
    receive
        stop -> void;
        {From, {swap_code, F1}} ->
            From ! {Name, ok, ack},
            loop(Name, F1, State);
        {From, Query} ->
            case (catch F(Query, State)) of
                {'EXIT', Why} ->
                    log_error(Name, Query, Why),
                    From ! {Name, crash},
                    loop(Name, F, State);
                {Reply, State1} ->
                    From ! {Name, ok, Reply},
                    loop(Name, F, State1)
            end
    end.

log_error(Name, Query, Why) ->
    io:format("Server ~p query ~p caused exception ~p~n",
              [Name, Query, Why]).
```

Figure 4.7: A simple server with error recovery and dynamic code replacement.

```
6> server3:swap_code(vshlr,
     fun(I,J) -> vshlr1:handle_event(I, J) end).
ok
7> vshlr3:find("robert").
{ok,"FMV"}
```

The above trace illustrates how to change the code in the server "on the fly." Lines 1-3 show the server working properly. In line 4 we trigger the deliberate error programmed into `vshlr3`. Server3 handles this error without crashing, so, for example, the response in line 5 is correct. In line 6 we send a command to change the code in the server back to the correct version in `vshlr1`. After this command has completed, the server works correctly as shown in line 7.

The programmer who wrote the code in `vshlr3` did not need to know anything about how `server3` was implemented nor that the code in the server could be dynamically changed without taking the server out of service.

The ability to change software in a server without stopping the server partially fulfils requirement 8 on page 14—namely to upgrade the software in a system without stopping the system.

If we return once again to the code for `server3` in figures 4.5 (the server) and `vshlr2` in figure 4.6 (the application), we observe the following:

1. The code in the server can be re-used to build many different client-server applications.

2. The application code is much simpler than the server code.

3. To write the server code the programmer must understand all the details of Erlang's concurrency model. This involves name registration, process spawning, untrappable exits to a process, and sending and receiving messages. To trap an exception the programmer must understand the notion of an exception and be familiar with Erlang's exception handling mechanisms.

4. To write the application, the programmer only has to understand how to write a simple sequential program—they need to know nothing about concurrency or error handling.

5. We can imagine using the *same* application code in a succession of progressively more sophisticated servers. I have shown three such servers but we can imagine adding more and more functions to the server while keeping the server/application interface unchanged.

6. The different servers (`server1`, `server2` etc) imbue the application with different non-functional characteristics. The functional characteristics for all servers are the same (that it, given correctly typed arguments all programs will eventually produce identical results); but the non-functional characteristics are different.

7. The code which implements the non-functional parts of the system is limited to the server (by non-function we mean things like how the system behaves in the presence of errors, how long time function evaluation takes, etc) and is hidden from the application programmer.

8. The details of how the remote procedure call is implemented are hidden inside the server module. This means that the implementation could be changed at a later stage without changing the client code, should this become necessary. For example, we could change the details of how `rpc/2` is implemented in figure 4.5 without having to change any of the client software which calls the functions in `server2`.

The division of the total server functionality into a non-functional part and a functional part is good programming practice and gives the system several desirable properties, some of which are:

1. Concurrent programming is often perceived as being difficult. In a large programming group, where the programmers have different skill levels the expert programmers should write the generic server

code, and the less-experienced programmers should write the application code.

2. Formal methods could be applied to the (simpler) application code. Work on the formal verification of Erlang, or on type systems designed to infer types from Erlang code have problems analysing concurrent programs. If the code in the generic servers is *assumed* to be correct *a priori* then the problem of proving properties of the system reduces to the problem of proving properties of sequential programs.

3. In a system with a large number of client-servers all the servers can be written with the *same* generic server. This makes it easier for a programmer to understand and maintain several different servers. In section 8.3.1 we will investigate this claim when we analyse a large system with many servers.

4. The generic servers and applications can be tested separately and independently. If the interface remains constant over a long period of time then both can be independently improved.

5. The application code can be "plugged into" a number of different generic servers, where these servers have different non-functional characteristics. Special servers, with the same interface, could provide an enhanced debugging environment etc. Other servers could provide clustering, hot standby etc. This has been done in a number of projects, for example Eddie [31] provided clustering and the Bluetail mail robustifier [11] provided a server with hot standby facilities.

4.2 Maintaining the Erlang view of the world

The Erlang view of the world is that *everything is a process* and that processes can only interact by exchanging messages.

When we interface Erlang programs to external software it is often convenient to write an interface program which maintains the illusion that "everything is a process."

As an example of this, we consider how to implement a web-server. Web-servers communicate with clients using the HTTP protocol as defined in RFC2616 [36].

From the point of view of an Erlang programmer, the inner loop of a web server would spawn a new process for each connection, accept a request from the client and respond appropriately. The code for this would be something like:

```
serve(Client) ->
  receive
    {Client, Request} ->
      Response = generate_response(Request)
      Client ! {self(), Response}
  end.
```

Where `Request` and `Response` are Erlang terms representing HTTP requests and HTTP responses.

The above server is very simple, it expects a single request, replies with a single response and terminates the connection.

A more sophisticated server, would support HTTP/1.1 persistent connections, the code for this is remarkably simple:

```
serve(Client) ->
  receive
    {Client, close} ->
      true;
    {Client, Request} ->
      Response = generate_response(Request)
      Client ! {self(), Response},
      server(Client);
    after 10000 ->
      Client ! {self(), close}
  end.
```

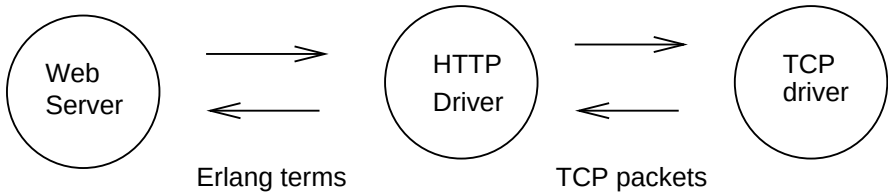


Figure 4.8: A web server

This 11 line function is essentially all that is needed to make a primitive web server with persistent connections.

The web server does not talk directly to the clients which are making HTTP requests. To do so would significantly clutter up the web-server code with irrelevant detail and make the structure difficult to understand.

Instead it makes use of a "middle-man" process (see figure 4.8). The middle-man process (an HTTP driver) converts between HTTP requests and responses and the corresponding Erlang terms which represent these requests and responses.

The overall structure of the HTTP driver is as follows:

```
relay(Socket, Server, State) ->
    receive
        {tcp, Socket, Bin} ->
            case parse_request(State, Data) of
                {completed, Request, State1} ->
                    Server ! {self(), {request, Req}},
                    relay(Socket, Server, State1);
                {more, State1} ->
                    relay(Socket, Server, State1)
            end;
        {tcp_closed, Socket} ->
            Server ! {self(), close};
        {Server, close} ->
            gen_tcp:close(Socket);
        {Server, Response} ->
            Data = format_response(Response),
```

```
            gen_tcp:send(Socket, Data),
            relay(Socket, Server, State);
        {'EXIT', Server, _} ->
            gen_tcp:close(Socket)
    end.
```

If a packet comes from the client, via a TCP socket, it is parsed by calling `parse_request/2`. When the response is complete an Erlang term representing the request is sent to the server. If a response comes from the server it is reformatted and then sent to the client. If either side terminates the connection, or an error occurs in the server, the connection is closed down. If this process terminates for any reason all the connections are automatically closed down.

The variable `State` is a state variable representing the state of a re-entrant parser that is used to parse the incoming HTTP requests.

The entire code for the web-server is not shown here but can be downloaded from [15].

4.3 Error handling philosophy

Error handling in Erlang is radically different to error handing in most other programming languages. The Erlang philosophy for handling errors can be expressed in a number of slogans:

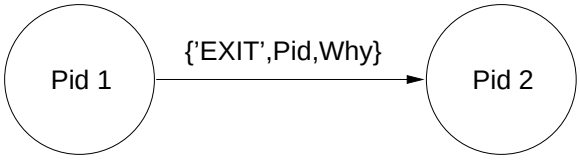
- Let some other process do the error recovery.
- If you can't do what you want to do, die.
- Let it crash.
- Do not program defensively.

4.3.1 Let some other process fix the error

How can we handle errors in a distributed system? To handle hardware errors we need replication. To guard against the failure of an entire computer we need two computers.

If computer 1 fails, computer 2 notices the failure and corrects the error.

If the first computer crashes the failure is noticed by the second computer which will try to correct the error. We use exactly the same method in Erlang, only instead of computers we use pairs of processes.



If process 1 fails, process 2 notices the failure and corrects the error.

If the process `Pid1` fails and if the processes `Pid1` and `Pid2` are linked together and if process `Pid2` is set to trap errors then a message of the form `{'EXIT',Pid1,Why}` will be delivered to `Pid2` if `Pid1` fails. `Why` describes the reason for failure.

Note also that if the computer on which `Pid1` executes dies, then an exit message `{'EXIT',Pid1,machine_died}` will be delivered to `Pid2`. This message *appears* to have come from `Pid1`, but in fact comes from the run-time system of the node where `Pid2` was executing.

The reason for coercing the hardware error to make it look like a software error is that we don't want to have two different methods for dealing with errors, one for software errors and the other for hardware errors. For reasons of conceptual integrity we want one uniform mechanism. This, combined with the extreme case of hardware error, and the failure of entire processors, leads to the idea of handling errors, not where they occurred, but at some other place in the system.

Thus under all circumstances, including hardware failure it is `Pid2`'s job to correct the error. This is why I say *"let some other process fix the error."*

This philosophy is completely different to that used in a sequential programming language where there is no alternative but to try and handle all errors in the thread of control where the error occurs. In a sequential language with exceptions, the programmer encloses any code that is likely to fail within an exception handling construct and tries to contain all errors that can occur within this construct.

Remote handling of error has several advantages:

1. The error-handling code and the code which has the error execute within different threads of control.

2. The code which solves the problem is not cluttered up with the code which handles the exception.

3. The method works in a distributed system and so porting code from a single-node system to a distributed system needs little change to the error-handling code.

4. Systems can be built and tested on a single node system, but deployed on a multi-node distributed system without massive changes to the code.

4.3.2 Workers and supervisors

To make the distinction between processes which perform work, and processes which handle errors clearer we often talk about *worker* and *supervisor* processes:

One process, the *worker* process, does the job. Another process, the *supervisor* process. observes the worker. If an error occurs in the worker, the supervisor takes actions to correct the error. The nice thing about this approach is that:

1. There is a clean separation of issues. The processes that are supposed to do things (the workers) do not have to worry about error handling.

2. We can have special processes which are only concerned with error handling.

3. We can run the workers and supervisors on *different* physical machines.

4. It often turns out that the error correcting code is *generic*, that is, generally applicable to many applications, whereas the worker code is more often application specific.

Point three is crucial—given that Erlang satisfies requirements R3 and R4 (see page 27) then we can run worker and supervisor processes on different physical machines, and thus make a system which tolerates hardware errors where entire processes fail.

4.4 Let it crash

How does our philosophy of handling errors fit in with coding practices? What kind of code must the programmer write when they find an error? The philosophy is *let some other process fix the error*, but what does this mean for their code? The answer is *let it crash*. By this I mean that in the event of an error, then the program should just crash. But what is an error? For programming purpose we can say that:

- *exceptions* occur when the run-time system does not know what to do.

- *errors* occur when the programmer doesn't know what to do.

If an exception is generated by the run-time system, but the programmer had foreseen this and knows what to do to correct the condition that caused the exception, then this is not an error. For example, opening a file which does not exist might cause an exception, but the programmer might decide that this is not an error. They therefore write code which traps this exception and takes the necessary corrective action.

Errors occur when the programmer does not know what to do. Programmers are supposed to follow specifications, but often the specification does not say what to do and therefore the programmer does not know what to do. Here is a example:

Suppose we are writing a program to produce code for a microprocessor, the specification says that a load operation is to result in opcode 1 and a store operation should result in opcode 2. The programmer turns this specification into code like:

```
asm(load)  -> 1;
asm(store) -> 2.
```

Now suppose that the system tries to evaluate `asm(jump)`—what should happen? Suppose you are the programmer and you are used to writing defensive code then you might write:

```
asm(load)  -> 1;
asm(store) -> 2;
asm(X)     -> ??????
```

but what should the ??????'s be? What code should you write? You are now in the situation that the run-time system was faced with when it encountered a divide-by-zero situation and you cannot write any sensible code here. All you can do is terminate the program. So you write:

```
asm(load)  -> 1;
asm(store) -> 2;
asm(X)     -> exit({oops,i,did,it,again,in,asm,X})
```

But why bother? The Erlang compiler compiles

```
asm(load)  -> 1;
asm(store) -> 2.
```

almost as if it had been written:

```
asm(load)  -> 1;
asm(store) -> 2;
asm(X)     -> exit({bad_arg, asm, X}).
```

The defensive code *detracts* from the pure case and confuses the reader–the diagnostic is often no better than the diagnostic which the compiler supplies automatically.

4.5 Intentional programming

Intentional programming is a name I give to a style of programming where the reader of a program can easily see what the programmer intended by their code. The intention of the code should be obvious from the names of the functions involved and not be inferred by analysing the structure of the code. This is best explained by an example:

In the early days of Erlang the library module `dict` exported a function `lookup/2` which had the following interface:

```
lookup(Key, Dict) -> {ok, Value} | notfound
```

Given this definition `lookup` was used in three different contexts:

1. For *data retrieval*–the programmer would write:

   ```
   {ok, Value} = lookup(Key, Dict)
   ```

   here lookup is used for to extract an item with a known key from the dictionary. `Key` should be in the dictionary, it is a programming error if this is not the case, so an exception in generated if the key is not found.

2. For *searching*–the code fragment:

   ```
   case lookup(Key, Dict) of
       {ok, Val} ->
   ```

```
            ... do something with Val ...
        not_found ->
            ... do something else ...
    end.
```

searches the dictionary and we do not know if `Key` is present or not—it is not a programming error if the key is not in the dictionary.

3. For *testing the presence of a key*—the code fragment:

```
case lookup(Key, Dict) of
    {ok, _} ->
      ... do something ...
    not_found ->
      ... do something else ...
end.
```

tests to see if a specific key `Key` is in the dictionary.

When reading thousands of lines of code like this we begin to worry about *intentionality*—we ask ourselves the question "what did the programmer intend by this line of code?"—by analysing the above three usages of the code we arrive at one of the answers *data retrieval*, a *search* or a *test*.

There are a number of different contexts in which keys can be looked up in a dictionary. In one situation a programmer knows that a specific key should be present in the dictionary and it is a programming error if the key is not in the dictionary and the program should terminate. In another situation the programmer does not know if the keyed item is in the dictionary and their program must allow for the two cases where the key is present or not.

Instead of guessing the programmer's intentions and analyzing the code, a better set of library routines is:

```
dict:fetch(Key, Dict) = Val | EXIT
dict:search(Key, Dict) = {found, Val} | not_found.
dict:is_key(Key, Dict) = Boolean
```

Which precisely expresses the intention of the programmer–here no guesswork or program analysis is involved, we clearly read what was intended.

It might be noted that `fetch` can be implemented in terms of `search` and vice versa. If `fetch` is assumed primitive we can write:

```
search(Key, Dict) ->
    case (catch fetch(Key, Dict)) of
        {'EXIT', _} ->
            not_found;
        Value ->
            {found, Value}
    end.
```

This is not really good code, since first we generate an exception (which should signal that the program is in error) and then we correct the error.

Better is:

```
find(Key, Dict) ->
    case search(Key, Dict) of
        {ok, Value} ->
            Value;
        not_found ->
            exit({find, Key}
    end.
```

Now precisely one exception is generated which represents an error.

4.6 Discussion

Programming is a disciplined activity. Writing clear intentional code with apparent structure is difficult. Part of this difficulty has to do with choosing the right abstractions. To master a complex situation we use the method of "divide and conquer," we split complex problems into simpler sub-problems and then solve the sub-problems.

This chapter has shown how to split a number of complex problems into simpler sub-problems. When it comes to error-handling I have shown how to "abstract out" the errors, and argued that the program should be divided into "pure" code and code which "fixes the errors."

In writing a server, I have shown how to abstract out two non-functional properties of the server. I have shown how to write a version of a server which does not crash when there is an error in the characteristic function which defines the behaviour of the server and I have shown how the behaviour of the server can be changed without stopping the server.

Recovering from errors, and changing code in a running system are typical non-functional properties that many real systems are required to have. Often, programming languages and systems strongly support writing code with well-defined functional behaviour, but have poor support for the non-functional parts of the program.

In most programming languages it is easy[2] to write a pure function, whose value depends in a deterministic manner on the inputs to the function, but it is much more difficult, and sometimes impossible, to do things like changing the code in the system, or handling errors in a generic manner, or protecting one programmer's code from failures in another part of the system. For this reason the programmer makes use of services offered by the operating system—the operating system provides protection zones, concurrency etc usually in the guise of processes.

There is a sense in which the operating system provides "what the programming language designer forgot." In a language like Erlang an operating system is hardly necessary, indeed the OS just provides Erlang with a number of device drivers, none of the OS mechanisms for processes, message passing, scheduling, memory management etc are needed.

The trouble with using an OS to fill in for deficiencies in the programming language is that the underlying mechanisms of the operating system itself cannot easily be changed. The operating systems' ideas of what a process is or how to do interprocess scheduling cannot be changed.

By providing the programmer with lightweight processes, and primitive mechanisms for detecting and handling errors an application programmer

[2]Well it should be easy.

can easily design and implement their own application operating system, which is specifically designed for the characteristics of their specific problem. The OTP system, which is just an application program written in Erlang, is an example of this.

5 Programming Fault-tolerant Systems

Designers devote half the software in telephone switches to error detection and correction[48]
Richard Kuhn, National Institute of Standards and Technology

What is a fault-tolerant system and how can we program it? This question is central to this thesis and to our understanding of how to build fault-tolerant systems. In this chapter we define what we mean by "fault-tolerance" and present a specific method for programing fault-tolerant systems. We start with a couple of quotations:

> *We say a system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults.* — [16]
>
> ...
>
> *To design and build a fault-tolerant system, you must understand how the system should work, how it might fail, and what kinds of errors can occur. Error detection is an essential component of fault tolerance. That is, if you know an error has occurred, you might be able to tolerate it by replacing the offending component, using an alternative means of computation, or raising an exception. However, you want to avoid adding unnecessary complexity to enable fault tolerance because that complexity could result in a less reliable system.* — Dugan quoted in Voas [67].

The presentation here follows Dugan's advice, I explain exactly what happens when an abnormal condition is detected and how we can make a software structure which detects and corrects errors.

The remainder of this chapter describes:

- *A strategy for programming fault-tolerance* — the strategy is to fail immediately if you cannot correct an error and then try to do something that is simpler to achieve.
- *Supervision hierarchies* — these are hierarchical organisations of tasks.
- *Well-behaved functions* — are functions which are supposed to work correctly. The generation of an exception in a well-behaved function is interpreted as a failure.

5.1 Programming fault-tolerance

To make a system fault-tolerant we organise the software into a *hierarchy of tasks* that must be performed. The highest level task is to run the application according to some specification. If this task cannot be performed then the system will try to perform some simpler task. If the simpler task cannot be performed then the system will try to perform an even simpler task and so on. If the lowest level task in the system cannot be performed then the system will fail.

This method is intuitively attractive. It says *if we cannot do what we want to do, then try to do something simpler*. We also try to organise the software so that simpler tasks are performed by simpler software, so that the likelihood of success increases as the tasks become simpler.

As the tasks become simpler, the emphasis upon what operation is performed changes—we become more interested in protecting the system against damage than in offering full service. At all stages our goal is to offer an acceptable level of service though we become less ambitious when things start to fail.

When failures occur we become interested in protecting the system, and logging the precise reason for failure, so that we can do something

about the failure in the future. This implies that we need some kind of stable error log which will survive a crash. In exceptional circumstances our system might fail, but when this does happen we should never lose the information which tells us why the system failed.

To implement our hierarchy of tasks we need some precise notion of what is meant by the world "failure." Evaluating a function in Erlang can result in an exception. But exceptions are not the same as errors, and all errors do not necessarily lead to failures. So we need to discuss the distinction between exceptions, errors and failures.

The distinctions between exceptions, errors and failures largely has to do with where in the system an abnormal event is detected, how it is handled and how it is interpreted. We trace what happens when an abnormal situation occurs in the system—this description is "bottom up" ie it starts at the point in time where the error is detected.

- At the lowest level in the system the Erlang virtual machine detects an internal error—it detects a divide by zero condition, or a pattern matching error or something else. The important point about all of these detected conditions is that it is pointless to continue evaluating code in the processes where the error occurred. Since the virtual machine emulator cannot continue, it does the only thing possible and throws an exception.

- At the next level, the exception may or may not be caught. The program fragment which traps the exception may or may not be able to correct the error which caused the exception. If the error is successfully corrected, then no damage is done and the process can resume as normal. If the error is caught, but cannot be corrected, yet another exception might be generated, which may or may not be trapped within the process where the exception occurred.

- If there is no "catch handler" for an exception[1] then the process itself will fail. The reason for failure will be propagated to any processes which are currently linked to the process which has failed.

[1] ie the current function is not evaluating within the scope of a catch statement.

- The linked processes which receive these failure signals may or may not intercept and process these signals as if they were normal inter-process messages.

Thus we see that what starts off as an abnormal condition in the virtual machine emulator propagates upwards in the system. At every point in the upwards propagation of the error an attempt might be made to correct the error. This attempt might succeed or fail, thus we have very fine grain control over how and where we can handle errors.

A "corrected" error, that is, a situation which has been foreseen, and for which corrective code has successfully been executed, is not considered a fault.

So far we have just seen how abnormal conditions, lead to exceptions, how exceptions can be trapped, and how untrapped exceptions can lead to process failures, and how process failures can be detected by other processes in the system. These are the available mechanisms with which we can implement our "hierarchy of tasks."

5.2 Supervision hierarchies

Recall that at the beginning of this chapter we talked about the idea of a hierarchy of tasks. The basic idea is:

1. Try to perform a task.
2. If you cannot perform the task, then try to perform a simpler task.

To each task we associate an *supervisor process*—the supervisor will assign a *worker* to try and achieve the goals implied by the task. If the worker process fails with a non-normal exit then the supervisor will assume that the task has failed and will initiate some error recovery procedure. The error recovery procedure might be to restart the worker or failing this try to do something simpler.

Supervisors and workers are arranged into hierarchical trees, according to the following:

1. *Supervision trees* are trees of *Supervisors*.
2. *Supervisors* monitor *Workers* and *Supervisors*.
3. *Workers* are instances of *Behaviours*.
4. *Behaviours* are parameterised by *Well-behaved functions*.
5. *Well-behaved functions* raise exceptions when errors occur.

Where:

- *Supervision trees* are hierarchical trees of supervisors. Each node in the tree is responsible for monitoring errors in its child nodes.
- *Supervisors* are processes which monitor other processes in the system. The things that are monitored are either supervisors or workers. Supervisors must be able to detect exceptions generated by the things they are monitoring, and be able to start, stop and restart the things they are monitoring.
- *Workers* are processes which perform tasks.

  If a worker process terminates with a non-normal exit (see page 74) the supervisor will assume that an error has occurred and will take action to recover from the error.

  Workers in our model are not arbitrary processes, but are instances of one of a small number of generic processes (called *behaviours*).
- *Behaviours* are generic processes whose operation is entirely characterised by a small number of callback functions. These functions must be instances of *well-behaved functions*.

  An example of a behaviour is the the `gen_server` behaviour which is used for programming a distributed fault-tolerant client-server. This behaviour is parameterised by a number of WBFs.

  All the programmer has to understand in order to program a fault-tolerant distributed client-server is how to write a WBF. The client-server behaviour provides a fault-tolerant framework for concurrency

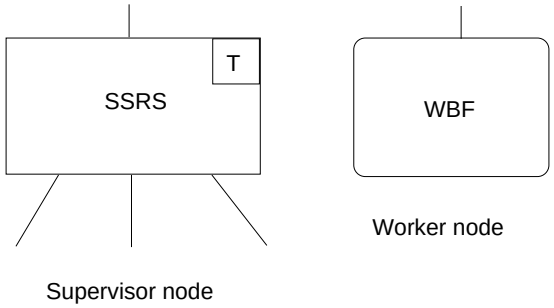


Figure 5.1: Supervisor and worker symbols

and distribution. The programmer need only be concerned with writing WBFs which parameterise this behaviour.

For convenience two particular supervision hierarchies are considered, namely *linear hierarchies* and *AND/OR hierarchy trees*. These are shown diagrammatically in the following sections.

5.2.1 Diagrammatic representation

Supervisors and workers can conveniently be represented using the notation in figure 5.1.

Supervisors are draw as rectangles with right-angled corners. In the top right-hand corner there is a symbol `T` denoting the type of the supervisor. The value of `T` will be one of `O` meaning "or" supervision, or `A` meaning "and" supervision. These supervision types are described later.

Supervisors can supervise an arbitrary number of workers or supervisors. For each entity that is supervised, the supervisor must know how to start, stop and restart the entity that it has to supervise. This information is kept in a `SSRS` which stands for "Start Stop and Restart Specification." The figure on page 148 contains a simple SSRS which specifies how three different behaviours are to be supervised.

Each supervisor (apart from the topmost supervisor in a hierarchy) has exactly one supervisor directly above it in the hierarchy, we call this the *Parent* of the supervisor. Conversely, processes below a given supervisor

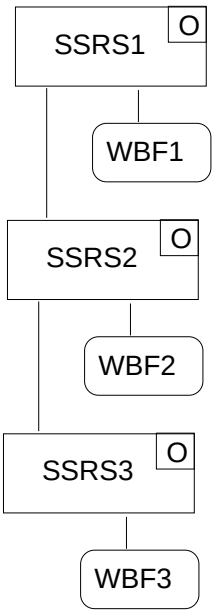


Figure 5.2: A linear supervisor hierarchy

in the hierarchy are referred to as the *children* of the supervisor. Figure 5.1 shows a supervisor node with one parent and three children.

Workers are drawn as rectangles with rounded corners (see Figure 5.1). They are parameterised by well-behaved functions (`WBF`s in the diagrams).

5.2.2 Linear supervision

I start with a linear hierarchy. Figure 5.2 represents a linear hierarchy of three supervisors. Each supervisor has a SSRS for each of its children and obeys the following rules:

- If my parent stops me then I should stop all my children.
- If any of my children dies then I must try to restart that child.

The system is started by starting the topmost supervisor. The top level supervisor is started first using the `SSRS1` specification. The top level supervisor has two children, a worker and a supervisor. It starts the worker

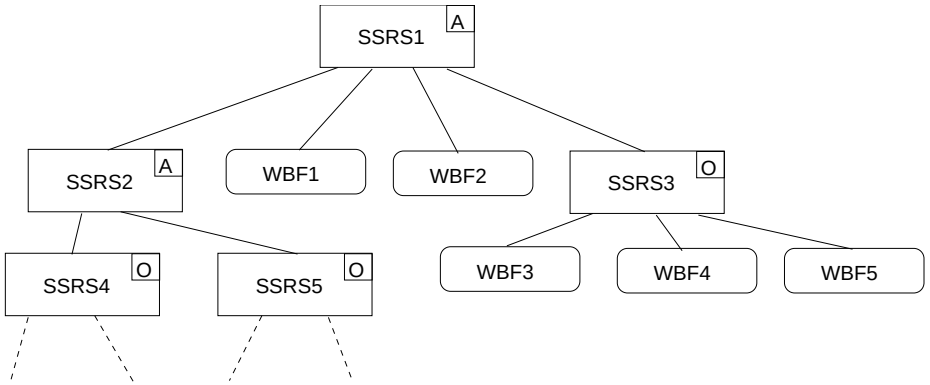


Figure 5.3: An AND/OR hierarchy

(which is a behaviour) parameterising it with the well-behaved function `WBF1` and it also starts the child supervisor. The lower level supervisors in the hierarchy are started in a similar manner until the entire system is running.

5.2.3 And/or supervision hierarchies

We can extend our simple supervision hierarchy into a tree structure containing only AND or OR nodes. Figure 5.3 shows such a tree. The denotation `A` denotes an AND supervisor, and `O` denotes an OR supervisor. The rules for a supervisor in an AND/OR tree are as follows:

- If my parent stops me then I should stop all my children.
- If any child dies and I am an AND supervisor stop all my children and restart all my children.
- If any child dies and I am an OR supervisor restart the child that died.

AND supervision is used for *dependent, or co-ordinated processes*. In the AND tree the successful operation of the system depends upon the successful operation of all the children–thus if any child dies all the children should be stopped and then restarted.

OR supervision can be used to co-ordinate the activities of *independent processes*. In the OR tree the activities being performed are assumed to be independent thus the failure of one child has no consequence for the other children—in the event of a child failure only that child process is restarted.

In concrete terms our "hierarchy of tasks" is expressed in a "supervision hierarchy".

In our system we equate tasks with goals and goals with *invariants*[2]—we will say that the goal has been achieved if the invariant associated with the goal is not false. In most programs the assertion of an invariant corresponds to the statement that evaluation of a particular designated function did not yield an exception.

Similar work has been reported by Candea and Fox [22] who have made a system based on "recursively-restartable Java components"

Notice that we distinguish errors into two categories, *correctable* and *uncorrectable* errors. Correctable errors are errors in a component which have been detected and corrected by the component. Uncorrectable errors are errors which have been detected but for which no corrective procedure has been specified.

The above discussion is all rather vague, since we have never said what an error is, nor have we said how we can in practice distinguish between a correctable and an uncorrectable error.

The situation is complicated by the fact that the vast majority of specifications only specify what should happen if all the components in a system work as planned—specifications rarely specify what should be done if a particular error occurs. Indeed if a specification does say exactly what to do if a particular error occurs, then it could be argued that this not an error at all but rather a desired property of the system. This is a consequence of the rather vague way in which the word "error" is used.

5.3 What is an error?

When we run a program the run-time system has no idea as to what constitutes an error—it just runs code. The only indication that something

[2] An invariant is something that is always true.

might have gone wrong is when an exception is generated.

Exceptions are generated automatically by the run-time system when the run-time system cannot decide what to do. For example, when performing a division the run-time system might detect a "divide by zero" condition in which case it generates an exception, because it does not know what to do. A complete list of all the conditions which cause exceptions can be found in section 3.5.1.

Exceptions do not always correspond to errors. If, for example, a programmer has written code which can correctly deal with a "divide by zero" exception then the presence of such an exception may not necessarily indicate an error.

It is the programmer who decides if an exception corresponds to an error—in our system the programmer must explicitly say which functions in the system are expected to never generate exceptions.

Schneider gives a number of definitions of fault-tolerance in his 1990 ACM tutorial paper. In this paper he says:

> *A component is considered faulty once its behaviour is no longer consistent with its specification* — [61].

For our purposes we will define an error as a *deviation between the observed behaviour of a system and the desired behaviour of a system.* The desired behaviour is "the behaviour that the specification says the system should have."

The programmer must ensure that if the system behaves in a way that deviates from the specification, some kind of error recovery procedure is initiated, and that some record of this fact is recorded in a permanent error log, so that it can be corrected later.

Building real systems is complicated by the fact that we often don't have a complete specification—in this case the programmer has some general notion as to what constitutes a error, and what does not. In the absence of an explicit specification we need an implicit mechanism that corresponds to our intuitive idea that an error is "something that causes the program to crash."

In the OTP system the programmer is expected to write *Well-behaved functions* (WBFs)—well-behaved functions are used to parameterise the

OTP behaviours. These functions are evaluated by the code in the OTP behaviours. If the evaluation of a parameterisation function generates an exception, then this is *defined* to be an error, and an error diagnostic will be added to the error log.

At this point it is important to recall part of the inner loop of the generic server shown on page 94, it was:

```
loop(Name, F, State) ->
    receive
        ...

        {From, Query} ->
            case (catch F(Query, State)) of
                {'EXIT', Why} ->
                    log_error(Name, Query, Why),
                    From ! {Name, crash},
                    loop(Name, F, State);
                {Reply, State1} ->
                    From ! {Name, ok, Reply},
                    loop(Name, F, State1)
            end
    end.
```

The callback function `F` is called within a `catch` statement. If an exception `Why` is generated, then this is assumed to be an error, and an error message is added to the error log.

This is a very simple example, but it does illustrate the fundamental principle of error handling used in the OTP behaviours. For example, in the OTP `gen_server` behaviour, the programmer must write a callback module `M` which is used to parameterise the server. The module `M` must, among other things, export the callback function `handle_call/2` (an example of this is shown in line 23–29 of figure 6.1 on page 136)—this callback function must be a well-behaved function.

5.3.1 Well-behaved functions

A well-behaved function (WBF) is a function which should not normally generate an exception. If an exception is generated by the WBF then the exception will be interpreted as an error.

If an exception occurs while trying to evaluate the WBF the WBF should try to correct the condition which caused the exception. If an error occurs within a WBF which cannot be corrected the programmer should terminate the function with an explicit `exit` statement.

Well-behaved functions are written according to the following rules:

Rule: 1 — The program should be isomorphic to the specification.

The program should faithfully follow the specification. If the specification says something silly then the program should do something silly. The program must faithfully reproduce any errors in the specification.

Rule: 2 — If the specification doesn't say what to do raise an exception.

This is very important. Specifications often say what is to be done if something happens, but omit to say what to do if something else happens. The answer is "raise an exception." Unfortunately many programmers take this as an opportunity for creative guess-work, and try to guess what the designer ought to have said.

If systems are written in this manner then the observation of an exception will be synonymous with a specification error.

Rule: 3 — If the generated exceptions do not contain enough information to be able to isolate the error, then add additional helpful information to the exception.

When programmers write code they should ask themselves what information will be written into the error log if an error occurs. If the information is insufficient for debugging purposes then they should add enough information to the exception to allow the program to be debugged at a later stage.

Rule: 4 — Turn non-functional requirements into assertions (invariants) that can be checked at run-time. If the assertion is broken then raise an exception.

An example of this might concern termination of a loop—a programming error might cause a function to enter an infinite loop causing the function never to return. Such an error could be detected by requiring that certain functions terminate within a specified time, by checking this time and generating an exception if the function did not terminate within the specified time.

6 Building an Application

The previous chapter introduced a general model for programming fault-tolerant systems, and introduced the idea of a "supervision tree" which is used to monitor the behaviour of the system. This chapter moves from a general theory to the specific implementation of supervisors as implemented in the OTP system.

To illustrate the supervision principles I build a simple OTP application. This application has a single supervisor process which manages three worker processes. The three worker processes are instances of the `gen_server`, `gen_event` and `gen_fsm` behaviours.

6.1 Behaviours

Applications which use the OTP software are built from a number of "behaviours." Behaviours are abstractions of common programming patterns, which can be used as the building blocks for implementing systems in Erlang. The behaviours discussed in the remainder of this chapter are as follows:

- `gen_server` — this is used to build servers which are used in client-server models.
- `gen_event` — this is used for building event handlers. Event handlers are things like error loggers, etc. An event handler is something

which responds to a stream of events, it does not necessarily reply to the processes which send events to the handler.

- `gen_fsm` — this is used for implementing finite state machines.
- `supervisor` — this is used for implementing supervision trees.
- `application` — this is used as a container for packaging completed applications.

For each behaviour I will present the general principles involved, the specific details of the programming API and give a complete example of how to make an instance of the behaviour.

In OTP systems are built in the following hierarchical manner:

- *Releases* — releases are at the top of the hierarchy. A release contains all the information necessary to build and run a system. A release will consists of a software archive (packed in some form) and a set of procedures for installing the release. The process of installing a release can be highly complicated, since upgrade releases often have to be installed without stopping the target systems. An OTP release packages all this complexity into a single abstract unit. Internally a release consists of zero or more applications.
- *Applications* — applications are simpler than releases, they contain all the code and operating procedures necessary to run a single application, and not the entire system. When a release is composed of multiple applications then the system should have been structured in such a way as to ensure that the different applications are largely independent from each other, or that that the different applications have strictly hierarchical dependencies.
- *Supervisors* — OTP applications are commonly built from a number of instances of supervisors.
- *Workers* — OTP supervisors supervise worker nodes. Worker nodes are usually instances of the `gen_server`, `gen_event` or `gen_fsm` behaviours.

Now for the application. The application is built bottom-up starting with the worker nodes. I will create three worker nodes (one instance of each of the gen_server, gen_event and gen_fsm behaviours.) The worker nodes are managed in a simple supervision tree, and the supervision tree is packaged into an application. I start with the worker nodes.

6.1.1 How behaviours are written

The OTP behaviours are written using a programming style which is similar to that used in the examples in section 4.1. There is however, one major difference. Instead of parameterising the behaviour with an arbitrary function, we parameterise the behaviour with the name of a module. This module must export a specific number of pre-defined functions. Exactly which functions must be exported depends upon the behaviour being defined. The complete APIs are documented in the behaviour manual pages.

As an example, assume that xyz is an instance of the gen_server behaviour, then xyz.erl has to contain code like the following:

```
-module(xyz).
-behaviour(gen_server).

-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, terminate/2, change_code/3]).
...
```

xyz.erl should export the six routines init/1 ... shown above. To create an instance of a gen_server we call:

```
gen_server:start(ServerName, Mod, Args, Options)
```

where ServerName names the server. Mod is the atom xyz. Args is an argument passed to xyz:init/1 and Options is an argument which controls the behaviour of the server itself. Options is never passed as an argument to the module xyz.

The method used for parameterising a behaviour in the examples shown on pages 86–101 is somewhat more general than the method used

in the OTP system. The reasons for the differences are largely historical, the first behaviours being written before funs were added to Erlang.

6.2 Generic server principles

Chapter 4 introduced the idea of a generic server. The generic server provides an "empty" server, that is, a framework from which instances of servers can be built. The examples in Chapter 4 were deliberately short, and illustrated the principles involved in making a generic server.

In the OTP system the Erlang module `gen_server` is used to make client-server modules. `gen_server` can be parameterised in a number of different ways to make a range of different types of servers.

6.2.1 The generic server API

To understand the `gen_server` API it is helpful to see the flow of control between the server and the application. I will describe the subset of the `gen_server` API that I have used in the examples in this chapter.

`gen_server:start(Name1,Mod,Arg,Options) -> Result` where

`Name1` = Name of server (see note 1).

`Mod` = Name of callback module (see note 3).

`Arg` = An argument which is passed to `Mod:init/1` (see note 4).

`Options` = A set of options which controls the workings of the server.

`Result` = The result obtained by evaluating `Mod:init/1` (see note 4).

`gen_server:call(Name2,Term) -> Result` where

`Name2` = Name of server (see note 2).

Term = Argument passed to Mod:handle_call/3 (see note 4).

Result = The result obtained by evaluating Mod:handle_call/3 (see note 4).

gen_server:cast(Name2,Term) -> ok where

Name2 = Name of server (see note2).

Term = An argument passed to Mod:handle_cast/3 (see note 4).

Notes:

1. Name1 is a term like {local,Name2} or {global,Name2}. Starting a local server creates a server on a single node. Starting a global server starts a server which can be transparently accessed from any node in a set of distributed Erlang nodes.

2. Name2 is an atom.

3. Mod should export some or all of the following functions: init/1, handle_call/3, handle_cast/3, terminate/2. These routines will be called by gen_server.

4. Some of the arguments to the functions in gen_server appear unchanged as the argument to the functions in Mod. Similarly, some of the terms contained in the return values from the functions in Mod reappear in the return values of the functions in gen_server.

The callback routines in Mod have the following APIs:

Mod:init(Arg) -> {ok,State}|{stop,Reason}

This routine attempts to start the server:

Arg
is the third argument supplied to gen_server:start/4.

`{ok,State}` means that starting the server succeeded. The internal state of the server becomes `State` and the original call to `gen_server:start` will return `{ok, Pid}` where `Pid` identifies the server.

`{stop,Reason}`
means starting the server failed in which case the original call to `gen_server:start` will return `{error,Reason}`.

`Mod:handle_call(Term,From,State) -> {reply,R,S1}`
This gets called when the user calls `gen_server:call(Name,Term)`.

`Term` is any term.

`From` identifies the client.

`State` is the current state of the server.

`{reply,R,S1}` causes the return value of `gen_server:call/2` to become `R` and the new state of the server to become `S1`.

`Mod:handle_cast(Term,State) -> {noreply, S1}|{stop,R,S1}`
This gets called when the user calls `gen_server:cast(Name,Term)`.

`Term` is any term.

`State` is the current state of the server.

`{noreply,S1}` causes the state of the server to change to `S1`.

`{stop,R,S1}` causes the server to stop. It will be stopped by calling `Mod:terminate(R,S1)`

`Mod:terminate(R, S) -> Void`

This routine gets called when the server stops. The return value is ignored.

`R` is the reason for termination.

`S` is the current state of the server.

6.2.2 Generic server example

This example makes a simple *Key–Value* server which is implemented using `gen_server`. The Key–Value server is implemented with a call-back module called `kv`[1] which is shown in figure 6.1.

Line 2 of `kv` tells the compiler that this module is a call-back module for the `gen_server` behaviour. The compiler will then issue warnings if the module does not export the correct set of call-back routines which are needed by `gen_server`.

`kv.erl` exports a number of client functions (line 4), and a number of callback functions (lines 6 and 7. The client functions can be called from anywhere within the system. The callback functions will only be called from within the `gen_server` module.

`kv:start()` starts the server by calling `gen_server:start_link/4`. The first argument to `gen_server:start_link/4` is the location of the server. In our example the location is `{local,kv}` which means that the server is a locally registered process with name `kv`. Several other values are permitted for the location. These include the value `{global,Name}` which would register the server with a global name (instead of a local name). Use of a global name allows the server to be accessed in a distributed Erlang system from any node in the system.

The remaining arguments to `gen_server:start/4` are the callback module name (`kv`), an initial argument (`arg1`) and a set of options controlling debugging etc ([]). Setting this to `[{debug,[trace,log]}]` would turn on debugging and tracing to a log file.

When `gen_server:start_link/4` is called it will initialise its internal data structures by calling `kv:init(Arg)` where `Arg` is the third argument that was supplied in the call to `gen_server:start_link/4`. For normal operation `init/1` should return a tuple of the form `{ok,State}`.

The client routines which are exported from `kv` in lines 18–21. `store/2` and `lookup/1` are implemented with `gen_server:call/2`.

Internally remote procedure calls are implemented by calling the call-back routine `handle_call/2`. Lines 23–29 implement the callback routines necessary to implement the server-side of the remote procedure calls.

[1] Line 26 of kv.erl has a deliberate error, ignore this for now.

```
-module(kv).
-behaviour(gen_server).

-export([start/0, stop/0, lookup/1, store/2]).

-export([init/1, handle_call/3, handle_cast/2,
         terminate/2]).

start() ->
    gen_server:start_link({local,kv},kv,arg1,[]).

stop()  -> gen_server:cast(kv, stop).

init(arg1) ->
    io:format("Key-Value server starting~n"),
    {ok, dict:new()}.

store(Key, Val) ->
    gen_server:call(kv, {store, Key, Val}).

lookup(Key) -> gen_server:call(kv, {lookup, Key}).

handle_call({store, Key, Val}, From, Dict) ->
    Dict1 = dict:store(Key, Val, Dict),
    {reply, ack, Dict1};
handle_call({lookup, crash}, From, Dict) ->
    1/0;  %% <- deliberate error :-)
handle_call({lookup, Key}, From, Dict) ->
    {reply, dict:find(Key, Dict), Dict}.

handle_cast(stop, Dict) -> {stop, normal, Dict}.

terminate(Reason, Dict) ->
    io:format("K-V server terminating~n").
```

Figure 6.1: A simple server

The first argument to handle_call is a pattern which must match the second argument used in the calls to gen_server:call/2. The second argument is the state of the server. In normal operation handle_call should return a tuple of the form {reply,R,State1} where R is the return value of the remote procedure call (which will be returned to the client, and become the return value of gen_server:call/2) and State1 which will become the new value of the state of the server.

gen_server:cast(kv,stop) which is called by stop/0 in line 12 is used to stop the server. The second argument stop reappears as the first argument to handle_cast/2 in line 31, the second argument is the state of the server. handle_cast returns {stop,Reason,State} which will force the generic server to call kv:terminate(Reason,State). This is to give the server a chance to perform any final operations that it wishes to perform before exiting. When terminate/2 returns, the generic server will be stopped and all name registrations removed.

In this example we have only shown a simple example of the use of the generic server. The manual pages for gen_server give all the options for the values that the various arguments to the callback functions and the control functions to gen_server can take. The generic server can be parameterised in a large number of different ways, making it easy to run the system as a local server or as a global server in a network of distributed Erlang nodes.

The generic server also has a number of in-built debugging aids which are automatically made available to the programmer. In the event of an error occurring in a server which has been built using gen_server a complete trace dump of what went wrong will automatically be added to the system error log. This information is usually sufficient to allow post-mortem debugging of the server.

6.3 Event manager principles

The event manager behaviour gen_event provides a general framework for building application-specific event handling routines. Event managers can be built for tasks like:

- error logging.
- alarm handling.
- debugging.
- equipment management.

Event managers provide named objects to which events can be sent. Zero or more event handlers can be installed *within* a given event manager.

When an event arrives at an event manager it will be processed by all the event handlers which have been installed within the event manager. Event managers can be manipulated at run-time. In particular we can install an event handler, remove an event handler or replace one event handler with a different handler.

We start with a few definitions:

- *Event* — something which happens.
- *Event Manager* — a program which coordinates processing of events of the same category. The event manager provides a named object to which events can be sent.
- *Notification* — the act of sending an event to an event manager.
- *Event Handler* — a function which can process events. The event handler must be a function of type:

  ```
  State x Event -> State'
  ```

The event manager maintains a list of module $\times$ state 2-tuples of the form `{M,S}`. We call such a list a *module – state* (MS) list.

Suppose that the internal state of the event manager is represented by the MS list:

```
[{M1, S1}, {M2, S2}, ...]
```

when an event `E` is received by the event manager it replaces this list by the list:

```
[{M1, S1New}, {M2, S2New}, ...].
```

where `{ok, SiNew} = Mi:handle_event(E, Si}`.

The event manager can be thought of as a generalisation of a conventional finite state machine, where instead of a single state, we maintain a "set" of states and a set of state transition functions.

As might be expected, there are also a number of interface functions in the `gen_event` API to allow manipulation of the `{Module,State}` pairs in the server. `gen_event` is more powerful than might be imagined from this simple introduction. The full details are best appreciated by reading the tutorial on event handling which is part of the OTP documentation.

6.3.1 The event manager API

The event manager (`gen_event`) exports the following routines:

`gen_event:start(Name1) -> {ok, Pid}|{error,Why}`

Create a new event manager.

`Name1` is the name of the event manager (see note 1).

`{ok,Pid}` means the event manager started successfully. `Pid` is the process Pid of the event manager.

`{error,Why}` is returned if an event manager cannot be started.

`gen_event:add_handler(Name2,Mod,Args) -> ok | Error`

Adds a new handler to the event manager. If the old state of the event manager was `L` then if this operation succeeds the new state of the event manager will be `[{Mod,S}|L]` where `S` is obtained by evaluating `Mod:init(Args)`.

`Name2` is the name of the event manager (see note 1).

`Mod` is the name of a callback module (see note 2).

`Arg` = An argument which is passed to `Mod:init/1`.

`gen_event:notify(Name2,E) -> ok`
: Sends an event `E` to the event manager. If the state of the event manager is a set of `{Mi,Si}` pairs and an event `E` is received then the state of the event manager changes to a set of `{Mi,SiNew}` pairs where `{ok,SiNew}=Mi:handle_event(E, Si)`

`gen_event:call(Name2,Mod,Args) -> Reply`
: Perform an operation on one of the handlers in the event manager. If the state of the event manager contains a tuple `{Mod,S}` then `Mod:handle_call(Args, S)` is called. `Reply` is derived from the return value of this call.

`gen_event:stop(Name2) -> ok`
: Stops the event manager.

Notes:

1. Event mangers follow the same naming conventions used for generic servers.
2. An event handler must export some of or all the following functions: `init/1`, `handle_event/2`, `handle_call/3`, `terminate/2`.

The event handler modules have the following API:

`Mod:init(Args) -> {ok,State}` where

`Args` comes from `gen_event:add_handler/3`.

`State` is the initial value of the state associated with this event handler.

`Mod:handle_event(E,S) -> {ok,S1}` where

`E` comes from the second argument to `gen_event:notify/2`.

`S` is the initial value of the state associated with this event handler.

S1 is the new value of the state associated with this event handler.

`Mod:handle_call(Args, State) -> {ok,Reply,State1}` where

Args comes from the second argument to `gen_event:call/2`.

State is the initial value of the state associated with this event handler.

Reply becomes the return value of `gen_event:call/2`

State1 is the new value of the state associated with this event handler.

`Mod:terminate(Reason, State) -> Void` where

Reason tells why the event manager is being stopped.

State is the current value of the state associated with this event handler.

6.3.2 Event manager example

Figure 6.2 shows how `gen_event` can be used to build a simple event logger. The error logger keeps track of the last five error messages, and it can display the last five error messages in response to notification of a `report` event.

Notice that the code in `simple_logger.erl` is purely sequential. At this point the observant reader should also have noticed the similarity between the forms of the arguments to `gen_server` and to `gen_event`. In general the arguments to routines like `start`, `stop`, `handle_call` etc in the different behaviour modules are chosen to be as similar as possible.

6.4 Finite state machine principles

Many applications (for example protocol stacks) can be modeled as finite state machines (FSMs). FSMs can be programmed using the finite state

```
-module(simple_logger).
-behaviour(gen_event).

-export([start/0, stop/0, log/1, report/0]).

-export([init/1, terminate/2,
         handle_event/2, handle_call/2]).

-define(NAME, my_simple_event_logger).

start() ->
    case gen_event:start_link({local, ?NAME}) of
      Ret = {ok, Pid} ->
        gen_event:add_handler(?NAME,?MODULE,arg1),
        Ret;
      Other ->
        Other
    end.

stop() -> gen_event:stop(?NAME).

log(E) -> gen_event:notify(?NAME, {log, E}).

report() ->
    gen_event:call(?NAME, ?MODULE, report).

init(arg1) ->
    io:format("Logger starting~n"),
    {ok, []}.

handle_event({log, E}, S) -> {ok, trim([E|S])}.

handle_call(report, S)    -> {ok, S, S}.

terminate(stop, _)        -> true.

trim([X1,X2,X3,X4,X5|_])  -> [X1,X2,X3,X4,X5];
trim(L)                   -> L.
```

Figure 6.2: A simple error logger

machine behaviour gen_fsm.

A FSM can be described as a set of rules of the form:

```
State(S) x Event(E) -> Actions (A) x State(S')
...
```

Which we interpret as meaning:

> If we are in the state S and the event E occurs we should perform the actions A and make a transition to the state S'.

If we choose to program a FSM using the gen_fsm behaviour then the state transition rules should be written as a number of Erlang functions which follow the following convention:

```
StateName(Event, StateData) ->
    .. code for actions here ...
    {next_state, StateName', StateData'}
```

6.4.1 Finite state machine API

The finite state machine behaviour (gen_fsm) exports the following routines:

`gen_fsm:start(Name1,Mod,Arg,Options) -> Result`
: This function works exactly like gen_server:start/4 which was discussed earlier.

`gen_fsm:send_event(Name1, Event) -> ok`
: Send an event to FSM identified by Name1

The callback module Mod must export the following functions:

`Mod:init(Arg) -> {ok,StateName,StateData}`
: When a FSM starts it calls init/1 this is expected to return an initial state StateName and some data associated with the state StateData. When gen_fsm:send_event(..., Event) is next called the FSM will evaluate Mod:StateName(Event,StateData).

`Mod:StateName(Event,SData) -> {nextstate,SName1,SData1}`
: Step the FSM. `StateName`, `Event` and `SData` represent the current state of the FSM. The next state of the FSM should be `SName1` and the data associated with the next state is `SData1`.

6.4.2 Finite state machine example

To illustrate a typical FSM application I have written a simple packet assembler using `gen_fsm`. The packet assembler is in one of two states `waiting` or `collecting`. When it is in the `waiting` state it expects to be sent information containing a packet length, in which case it enters the `collecting` state. In the `collecting` state it expects to be sent a number of small data packets, which it has to assemble. When the length of all the small data packets equals the total packet length it prints the assembled packet and re-enters the `waiting` state.

Figure 6.3 is a simple packet assembler written using `gen_fsm`. In line 11 we call `gen_fsm:start_link/4` to start a local instance of the FSM behaviour—Note the similarity to `gen_server:start_line/4` in Line 10 of Figure 6.1. The third argument to `gen_server:start/4` reappears is passed as an argument to `init/1` in line 17.

The `waiting` state is modeled by the function `waiting/2` (line 21) and the `collecting` state by `collecting/2` (lines 24–34). The data associated with the state is stored in the second argument to these functions. The first argument of both these functions is provided by the second argument in the calls made to `gen_fsm:send_event/2`. So, for example, `send_data/1` calls `gen_fsm:send_event/2` with a second argument `Len`. This argument reappears as the first argument to `waiting/2` in line 21.

The state of the FSM is represented by a 3-tuple `{Need,Len,Buff}`. When collecting data, `Need` is the total number of bytes that should be collected, `Len` is the number which have actually been collected, and `Buff` is a buffer containing the bytes which have actually been collected. This 3-tuple appears as the second argument to `collect/2` in line 24.

We can illustrate the use of the packet assembler in a session with the Erlang shell:

```
-module(packet_assembler).
-behaviour(gen_fsm).

-export([start/0, send_header/1, send_data/1]).

-export([init/1,terminate/3,waiting/2,collecting/2]).

-define(NAME, my_simple_packet_assembler).

start() ->
  gen_fsm:start_link({local, ?NAME},?MODULE,arg1,[]).

send_header(Len) -> gen_fsm:send_event(?NAME, Len).

send_data(Str)   -> gen_fsm:send_event(?NAME, Str).

init(arg1) ->
  io:format("Packet assembler starting~n"),
  {ok, waiting, nil}.

waiting(N, nil) ->
  {next_state, collecting, {N,0,[]}}.

collecting(Buff0, {Need, Len, Buff1}) ->
  L = length(Buff0),
  if
    L + Len < Need ->
      {next_state, collecting,
        {Need, Len+L, Buff1++Buff0}};
    L + Len == Need ->
      Buff = Buff1 ++ Buff0,
      io:format("Got data:~s~n", [Buff]),
      {next_state, waiting, nil}
  end.

terminate(Reason, State, Data) ->
  io:format("packet assembler terminated:"
            "~p ~n", [Reason]),
  true.
```

Figure 6.3: A simple packet assembler

```
> packet_assembler:start().
{ok,<0.44.0>}
> packet_assembler:send_header(9).
ok
> packet_assembler:send_data("Hello").
ok
> packet_assembler:send_data(" ").
ok
> packet_assembler:send_data("Joe").
Got data:Hello Joe
ok
```

Again `gen_fsm` has more functionality than can be described here.

6.5 Supervisor principles

Up to now we have concentrated on primitive behaviours which solve typical application problems, and a large part of an application can be written using the primitive client-server, event handling and FSM behaviours. The `gen_sup` behaviour is the first meta-behaviour which is used to glue together primitive behaviours into supervision hierarchies.

6.5.1 Supervisor API

The supervisor API is extremely simple:

`supervisor:start_link(Name1,Mod,Arg) -> Result`
starts a supervisor but calling `Mod:init(Arg)`

`Mod` must export `init/1`, where

`Mod:init(Arg) -> SupStrategy` `SupStrategy` is a term describing the supervision tree.

SupStrategy is a term describing how workers in a supervision tree should be started, stopped or restarted. I will not describe this in any detail here. The example which follows contains a simple supervision tree which is explained here. Complete detail can be found in the supervisor manual pages.

6.5.2 Supervisor example

The example in figure 6.4 monitors the three worker behaviours which were introduced earlier in this chapter. Recall that kv.erl included a deliberate error (Line 26 of figure 6.1) and that simple_logger.erl also had an error[2] (which I didn't mention). We will see what happens when these errors occur at run-time.

The module simple_sup.erl (Figure 6.4) defines the behaviour of the supervisor. It starts in line 7 by calling supervisor:start_link/3 — the calling conventions harmonise with the start and start_link functions exported by the other behaviours in the system. ?MODULE is a macro which expands to the current module name simple_sup. The final argument is set to nil. The supervisor method starts by calling init/1 in the specified callback module, with argument nil (the third argument to start_link/3).

init/1 returns a data structure defining the shape of the supervision tree and the strategy to be used. The term {one_for_one,5,1000} (Line 11) tell the supervisor to construct an "or" supervision tree (see page 122)— this is because the three activities being supervised are considered unrelated. The numbers 5 and 1000 specify a *restart frequency*—if the supervisor has to restart the processes which it is monitoring more than 5 times in 1000 seconds then it itself will fail.

There are three objects in our supervision tree, but I will only describe how the packet assembler is added to the supervision tree. The other two behaviours follow the same principles as for the packet assembler.

Lines 13–15 specify the behaviour of the packet assembler.

The first item in the tuple describes how the packet assembler should

[2]You did notice the error, I hope.

```
1  -module(simple_sup).
2  -behaviour(supervisor).
3
4  -export([start/0, init/1]).
5
6  start() ->
7    supervisor:start_link({local, simple_supervisor},
8                          ?MODULE, nil).
9  init(_) ->
10   {ok,
11    {{one_for_one, 5, 1000},
12     [
13      {packet,
14        {packet_assembler, start, []},
15        permanent, 500, worker, [packet_assembler]},
16      {server,
17        {kv, start, []},
18        permanent, 500, worker, [kv]},
19      {logger,
20        {simple_logger, start, []},
21        permanent, 500, worker, [simple_logger]}]}}.
```

Figure 6.4: A simple supervisor

be supervised. This starts on line 13. The atom `packet` is an arbitrary name (which must be unique for this instance of a supervisor) which can be used to refer to the node in the supervision tree.

Because the objects which are being supervised are themselves instances of OTP behaviours then plugging them into the supervision tree is simple. The next argument (line 14) is a 3-tuple `{M,F,A}` which will be called by the supervisor to start the specified processes. The supervisor calls `apply(M,F,A)` when it wants to start a supervised process.

The first argument `permanent` on line 15 says that the monitored process is to be a so-called "permanent" process. A permanent process will be automatically restarted by the supervisor if it fails.

In addition to specifying how to start a supervised process, the supervised process itself must be written in a particular manner. It must for example, be possible for the supervisor to request the supervised process to terminate in an orderly manner. To do this the supervised processes must obey the so-called "Shutdown protocol."

To terminate a worker process the supervisor calls `shutdown(P,How)` where `P` is the Pid of the worker and `How` determines how the worker is to be stopped. `shutdown` is defined as follows:

```
shutdown(Pid, brutal_kill) ->
    exit(Pid, kill);
shutdown(Pid, infinity) ->
    exit(Pid, shutdown),
    receive
        {'EXIT', Pid, shutdown} -> true
    end;
shutdown(Pid, Time) ->
    exit(Pid, shutdown),
    receive
        {'EXIT', Pid, shutdown} ->
            true
    after Time ->
        exit(Pid, kill)
    end.
```

If How is brutal_kill then the worker process is killed (see page 75).

If How is infinity then a shutdown signal is sent to the worker process which should respond with an {'EXIT',Pid,shutdown} message.

If How is an integer T when the process is given T milliseconds to terminate. If a {'EXIT',Pid,shutdown} is not received within T milliseconds then the process is unconditionally killed.

In line 15 of figure 6.4 the integer 500 is a "shutdown time" which is needed in the shutdown protocol. It says that if the supervisor wishes to stop a supervised processe then it is to allow the process up to 500 milliseconds to stop what it is doing.

The worker argument means that the supervised process is a worker process (recall from page 119 that a supervised process can be either a worker or a supervisor process) and [packet_assembler] is a list of all modules used by this behaviour (this argument is needed for synchronising code change operations).

Once everything has been defined, we can compile and run the supervisor. In the following transcript, I start a supervisor and trigger a number of errors in the supervised processes. The supervised processes die and are automatically restarted by the supervisor.

The first example shows what happens when an error occurs in the packet assembler. We start the supervisor, and check the Pid of the packet assembler.

```
1> simple_sup:start().
Packet assembler starting
Key-Value server starting
Logger starting
{ok,<0.30.0>}
2> whereis(my_simple_packet_assembler).
<0.31.0>
```

The printout shows that the servers have started.

Now we send a 3-byte length count, followed by 4 bytes of data:[3]

[3]This was the second deliberate error and I'm sure you have already noticed it!

```
3> packet_assembler:send_header(3).
ok
4> packet_assembler:send_data("oops").
packet assembler terminated:
  {if_clause,
    [{packet_assembler,collecting,2},
     {gen_fsm,handle_msg,7},
     {proc_lib,init_p,5}]}
ok
Packet assembler starting
=ERROR REPORT==== 3-Jun-2003::12:38:07 ===
** State machine my_simple_packet_assembler terminating
** Last event in was "oops"
** When State == collecting
**      Data  == {3,0,[]}
** Reason for termination =
** {if_clause,[{packet_assembler,collecting,2},
               {gen_fsm,handle_msg,7},
               {proc_lib,init_p,5}]}
```

This results in quite a lot of output. Firstly the packet assembler crashes, which accounts for the first error printout. Secondly, the supervisor detects that the packet assembler has crashed and so it restarts it—the message "`Packet assembler starting`" is printed when the process restarts. Finally there is a long and hopefully informative error message.

The error message contains information about the state of the FSM at the point when it crashed. It tells us that the state of the FSM was `collecting` and that the data associated with this state was the 3-tuple `{3,0,[]}` and that the event which caused the FSM to crash was `"oops"`. This information should be sufficient to debug the FSM machine.

In this case the output from the error logger is directed to standard output. In a production system the error logger would be configured to direct its output to stable storage. Analysis of the error log should be sufficient for post-mortem debugging of the system.

We can confirm that the supervisor has correctly restarted the packet

assembler; evaluating `whereis(my_simple_packet_assembler)` returns the Pid of the newly started packet assembler.

```
6> whereis(my_simple_packet_assembler).
<0.40.0>
7> packet_assembler:send_header(6).
ok
8> packet_assembler:send_header("Ok now").
Got data:Ok now
ok
```

In a similar manner we can evoke the deliberate error in the Key–Value server:

```
12> kv:store(a,1).
ack
13> kv:lookup(a).
{ok,1}
14> spawn(fun() -> kv:lookup(crash) end).
<0.49.0>
K-V server terminating
Key-Value server starting
15>
=ERROR REPORT==== 3-Jun-2003::12:54:10 ===
** Generic server kv terminating
** Last message in was {lookup,crash}
** When Server state == {dict,1,
                                16,
                                16,
... many lines removed ...

** Reason for termination ==
** {badarith,[{kv,handle_call,3},{proc_lib,init_p,5}]}

15> kv:lookup(a).
error
```

Note the `kv:lookup(crash)` must be evaluated in a temporary processes which is not linked to the query shell. This is because the supervisor was started with the call `supervisor:start_link/4` and is thus linked to the query shell. Evaluating `kv:lookup(crash)` directly in the shell will crash the supervisor, which is probably not what was intended.[4]

Note how the supervisor and pre-defined behaviours work *together*. The supervisor and the primitive behaviours were not designed in isolation, but were designed to complement each other.

The default behaviour is to provide as much helpful information as possible in the error log and to put the system into a safe state.

6.6 Application principles

We have now constructed three primitive servers and built them into a supervision tree; what remains is to build everything into an application. An application is a container for everything that is needed to deliver a particular application.

The way in which applications are programmed differs from the earlier behaviours. The earlier behaviours use callback modules, which must export a number of pre-defined functions.

Applications do not use callback functions but instead assume a specific organisation of files, directories and sub-directories in the file system. The most important part of the application is contained in the application descriptor file (a file with extension `.app`) which describes all the resources needed by the application.

6.6.1 Applications API

Applications are described using an application descriptor file. Application descriptor files have the extension `.app`. The MAN (4) manual page for an application defines the structure of a `.app` file to have the following structure:

[4] My original attempt at this failed, but Chandrashekhar Mullaparthi on the Erlang mailing list was kind enough to point out why my behaviours were behaving badly.

```
{application, Application,
  [{description,  Description},
   {vsn,          Vsn},
   {id,           Id},
   {modules,      [Module1, .., ModuleN]},
   {maxT,         MaxT},
   {registered,   [Name1, .., NameN]},
   {applications, [Appl1, .., ApplN]},
   {included_applications, [Appl1, .., ApplN]},
   {env,          [{Par1, Val1}, .., {ParN, ValN}]},
   {mod,          {Module, StartArgs}},
   {start_phases,
        [{Phase1, PhaseArgs1}, ..,
         {PhaseN, PhaseArgsN}]}]}.
```

All keys in the application association list are optional. If omitted, reasonable default values are used.

6.6.2 Application example

To package our application which consists of three primitive behaviours and one supervisor we use the application file `simple.app` shown in figure 6.5.

In our example the structure of the `.app` file is straight-forward.

The main purpose of the application files is to name and describe the application and to list all the modules and registered process names that the application uses.

In addition to the `simple.app` we need a main program which we use to "launch" the application; we can use `simple.erl` which in figure 6.6. `simple.erl` has a couple of calls to start and stop the application.

Now we are ready to run the application. Assuming that all the Erlang files are compiled and in the same directory as the `.app` file then we can start the application, and test one of the servers as follows:

```
1> application:start(simple, temporary).
Packet assembler starting
```

```
{application, 'simple',
 [{description, "A simple application"},
  {vsn,         "1.0"},
  {modules,      [simple,kv,packet_assembler,
                  simple_sup,simple_logger]},
  {maxT,         infinity},
  {registered,   [kv, my_simple_event_logger,
                  my_simple_packet_assembler]},
  {applications, []},
  {included_applications, []},
  {env,          []},
  {mod,          {simple, go}}]}.

```

Figure 6.5: simple.app - a simple application

```
-module(simple).
-behaviour(application).

-export([start/2).

start(_, _) -> simple_sup:start().

```

Figure 6.6: simple.erl - a simple application

```
Key-Value server starting
Logger starting
ok
2> packet_assembler:send_header(2).
ok
3> packet_assembler:send_data("hi").
ok
Got data:hi
```

Now we can stop the application:

```
4> application:stop(simple).

=INFO REPORT==== 3-Jun-2003::14:33:26 ===
    application: simple
    exited: stopped
    type: temporary
ok
```

After stopping the application *all* processes running within the application will be closed down in an orderly manner.

6.7 Systems and releases

The development of this chapter has been "bottom-up." I start with simple things, combining them into larger and more complicated units. I started with a number of primitive servers, `gen_server`, `gen_event` and `gen_fsm`. I organised these primitive behaviours into a supervision hierarchy, and then built the supervision hierarchy into an application.

The final stage, which is not shown here, is to build the application into a release. A release packages a number of different applications into a single conceptual unit. The result is small number of files which can be moved to a target environment.

Building a complete release is a complex procedure—not only must a release describe the current state of the system it must also know about previous versions of the system.

Releases contain not only information about the current version of the software but also information about previous releases of the software. In particular they contain procedures for upgrading the system from an earlier version of the software to the current version of the software. This upgrade must often be performed without stopping the system. A release must also handle the situation where the installation of the new software fails for some reason. If a new release fails, then the system should revert back to a previous stable state. All this is handled by the release management part of the OTP system.

When we look at the AXD301 project in chapter 8, we will see that there were 122 instances of `gen_server`, 36 instances of `gen_event` and 10 instances of `gen_fsm`. There were 20 supervisors and 6 applications. All this is packaged into one release.

In my opinion the simplest of these behaviours is `gen_server` which also happens to be the single most used design pattern. Errors in a `gen_server` callback module should result in informative error messages of sufficient quality to enable post-hoc debugging of the system.

Using the above behaviours is a compromise between design and programming efficiency. It is much quicker to design and program a system using design patterns, but the resultant code is marginally less efficient than hand-written code which solves exactly the same problem.

6.8 Discussion

- In the OTP system the generic modules which implemented the behaviours themselves were written by expert Erlang programmers. These modules are based on several years of experience and represent "best practice" in writing code for the particular problem.

- Systems built using the OTP behaviours have a very regular structure. For example, all client-servers and supervision trees have an identical structure. The use of the behaviour *forces* a common structure in the solution of the problem. The applications programmer has to provide that part of the code which defines the *semantics* of

their particular problem. All the *infrastructure* is provided automatically by the behaviour.

- It is relatively easy for a new programmer joining an existing team to *understand* behaviour-based solutions to problems. Once they have gained familiarity with the behaviours they can easily recognize situations where a particular behaviour should be used.

- Most of the "tricky" systems programming can be hidden within the implementation of the behaviours (which are actually much more complicated than described here). It you look back to the client-server and event handler behaviours you will see that all the code to do with concurrency, message passing etc is isolated in the "generic" part of the behaviour. The "problem specific" code only has pure sequential functions with well-defined types.

  This is a highly desirable state of affairs—concurrent programs which are "difficult" are isolated to small well-defined parts of the system. The vast majority of the code in the system can be written using sequential programs having well-defined types.

In our system the behaviours solve *orthogonal problems*—for example, client–server has nothing to do with worker-supervisor. In building a real system we pick and mix between behaviours and combine them in many different ways to solve problems.

Offering a small and fixed set of behaviours to a software designer has several benefits:

- It focuses attention on a small set of well-proven techniques. We know in advance that the individual techniques work well in practice. Given totally unconstrained choice and complete freedom of action in a design, the designer may be tempted to produce something which is unnecessarily complex or something which cannot be implemented.

- It allows the designer to structure and talk about the design in a precise manner. It provides a vocabulary for discourse.

- It completes the feedback cycle between design and implementation. All the behaviours presented here work in practice. They are all used, for example, in the Ericsson AXD301 product.

7 OTP

The Open Telecom Platform (OTP) is a development system designed for building and running telecommunications systems. A block diagram of the system is shown in figure 7.1, which is taken from the article in [66]. As can be seen from figure 7.1 the OTP system is a so-called "middleware platform" designed to be run on top of a conventional operating system.

The OTP system was developed internally at Ericsson. Most of the software was released into the public domain subject to the Erlang public license.[1]

Included in the OTP release are the following components:

1. Compilers and development tools for Erlang.
2. Erlang run-time systems for a number of different target environments.
3. Libraries for a wide range of common applications.
4. A set of design patterns for implementing common behavioural patterns.
5. Educational material for learning how to use the system.
6. Extensive documentation.

[1] Very similar to an open source license.

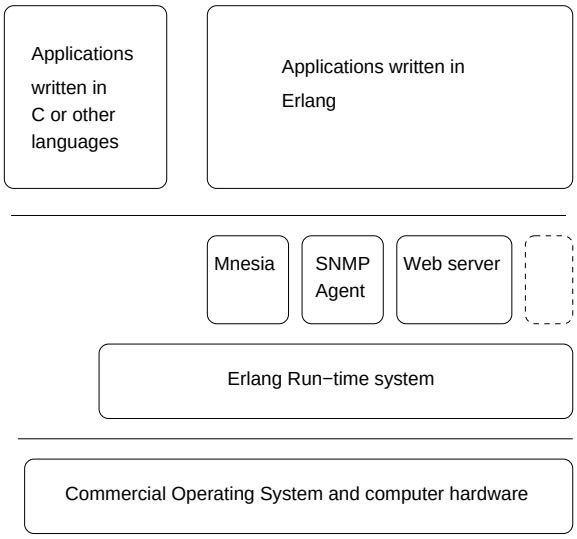


Figure 7.1: The OTP system architecture

OTP has been ported to a number of different operating systems, these include all the major Unix-like systems (Linux, FreeBSD, Solaris, OS-X ..), most of the Windows operating systems (Windows 95, 98, NT, ...) and a number of embedded operating systems like VxWorks.

The Erlang run-time system is a virtual machine designed to run the intermediate code produced by the Erlang BEAM compiler. It also provides run-time support services for a native code Erlang compiler.

The Erlang BEAM compiler replaced the original JAM compiler in 1998. The BEAM compiler [41, 42] compiles Erlang code into sequences of instructions for a 32-bit word threaded interpreter. The original JAM machine was a non-threaded byte code interpreter.

For additional efficiency Erlang programs can be compiled to native code using the HIPE compiler [47] developed at the University of Uppsala. Both interpreted BEAM and compiled code can be freely intermixed at a module level, ie, entire modules can be compiled to either BEAM or HIPE code, but code within an individual module cannot be intermixed.

Both the beam and HIPE machines use common code in the Erlang run-time system for memory management, input/output, process management, and garbage collection etc.

The Erlang run-time system offers many of the services which are traditionally offered by an operating system, so by comparison with the run-time support needed for a purely sequential language the run-time system is fairly complex. All Erlang processes are managed by the Erlang run-time system—even when there are several tens of thousands of Erlang processes running under control of the Erlang run-time system the host operating system will only think that there is one process running, that being the Erlang run-time system itself.

The Erlang compiler, on the other hand is rather simple, compared to most other languages. Compilation is often a simple translation of Erlang code into an appropriate primitive in the virtual machine. Thus, for example, the `spawn` primitive in Erlang compiles to a single opcode in the virtual machine (the spawn primitive) and great care was taken to implement this as efficiently as possible.

7.1 Libraries

The OTP release contains a large set of libraries, all of which for release purposes are considered instances of OTP applications. Release R9B contains the following applications:

- `appmon` – a graphical utility to observe and manipulate supervision trees.
- `asn1` – a compiler and run-time support for decoding and encoding packets defined in ASN.1.
- `compiler` – the Erlang compiler.
- `crypto` – a set of functions for encrypting and decrypting data and for computing message digests.
- `debugger` – an Erlang source code debugger.
- `erl_interface` – a set of libraries for communicating with distributed Erlang nodes.

- `erts` – the Erlang run-time system.
- `et` – the event tracer and tools to record and give a graphical presentation of event data.
- `eva` – the "event and alarm" handling application.
- `gs` – a graphics system. A set of graphics routines for building GUIs.
- `ic` – Erlang IDL compiler.
- `inets` – an HTTP server and an FTP client.
- `jinterface` – a tool to create Java to Erlang interfaces.
- `kernel` – one of the two basic libraries needed to run the system (the other is `stdlib`). `kernel` contains code for file servers, code servers etc.
- `megaco` – libraries for the Megaco[2]/H248 protocols.
- `mnemosyne` – a database query language for `mnesia`.
- `mnesia` – a distributed DBMS with soft real-time properties for Erlang.
- `observer` – tools for tracing and observing the behaviour of a distributed system.
- `odbc` – an Erlang interface ODBC interface to SQL databases.
- `orber` – an Erlang implementation of a CORBA object request broker. Note: there are also separate applications to provide access to various CORBA services, such as the events, notifications, file transfers etc.
- `os_mon` – a tool to monitor resource usage in the external operating system.

[2]Media Gateway Control.

- `parsetools` — tools for parsing Erlang. Includes `yecc` an LALR(1) parser generator.

- `pman` — a graphic tool to inspect the state of the system. Pman can be used to observe local or remote Erlang nodes.

- `runtime_tools` — miscellaneous small routines needed in the runtime system.

- `sasl` — short for "System Architecture Support Libraries." This application contains support for alarm handling, managing releases etc.

- `snmp` — an Erlang implementation of the Simple Network Management Protocol [24]. This application includes a MIB compilers and tools for building MIBs etc.

- `ssl` — an Erlang interface to the secure sockets layer.

- `stdlib` — the "pure" Erlang libraries needed to run the system. The other obligatory application is `kernel`.

- `toolbar` — a graphical toolbar from which applications can be started.

- `tools` — a package of stand-alone applications for analysing and monitoring Erlang programs. This includes tools for profiling, coverage analysis, cross reference analysis etc.

- `tv` — a "table viewer." The table viewer is a graphic application to allow graphic browsing of tables in the mnesia database.

- `webtool` — a system for managing web-based tools (such as `inets`).

The OTP libraries provide a highly sophisticated tool set and are a good starting point for any commercial product, they are however, fairly complex.

Recall that all of chapter 6 was devoted to a simplified explanation of five behaviours (`gen_server`, `gen_event`, `gen_fsm`, `supervisor` and `application`) and a complete explanation of any one of these behaviours is outside the scope of this thesis. The principles behind one of these behaviours (`gen_server`) was the subject of pages 86–101.

`stdlib` in release R9B has some 71 modules–four of them have been described here.

8 Case Studies

This section of the thesis presents studies of some systems which were written using the Erlang/OTP system. The first is the Ericsson AXD301 system—the AXD301 is a high-capacity ATM[1] switch. The version of the AXD301 system studied here has over 1.1 *million* lines of Erlang, which makes it one of the largest programs ever to be written in a functional style of programming. The AXD makes extensive use of the OTP libraries, so it provides a good test of the functionality of the libraries.

Following this I study a number of small products made by Bluetail AB or Alteon Web Systems/Nortel Networks. To avoid confusion, I might add that Bluetail AB was founded by most of the "Erlang people" who left the Ericsson CSLab (myself included) and that Bluetail was subsequently acquired by Alteon Web Systems, and that Nortel Networks then acquired Alteon. The products, however, were all developed by the same core group of people.

These products include the Bluetail Mail Robustifier (BMR) and the "SSL[2] accelerator," made by Alteon Web Systems and sold by Nortel Networks. The SSL accelerator was developed in a remarkably short time (9 months) and rapidly became the "market leader" in the niche market for "embedded secure socket layer devices." The SSL accelerator also makes extensive use of the Erlang/OTP system and libraries.

These projects represent two extremes. The AXD301 was done with

[1] Asynchronous Transfer Mode.

[2] Secure Socket Layer.

a large group of programmers; over 40 programmers have been involved with the code over a 4-year period. Large software projects are notoriously difficult to manage, and the resulting code is often difficult to understand; so one of the matters that I am concerned with is how well (or poorly) the OTP design methodology supports the construction of large systems.

The second set of projects was programmed by a much smaller group of programmers (5-10 depending upon the product) and was completed within a much shorter time frame (six months). The developers were all highly experienced Erlang programmers. Two of the developers, Magnus Fröberg and Martin Björklund, designed and programmed the original OTP behaviours. One of the developers, Claes Wikström was a co-author of the second edition of the Erlang book. Wikström was also the main implementor of distributed Erlang, and of the mnesia data base.

8.1 Methodology

In the case studies, I am interested in the following:

- *Problem domain* — what was the problem domain. Is this the kind of problem that Erlang/OTP was designed to solve?

- *Quantitative properties of the code* — how many lines of code were written? How many modules? How was the code organised? Did the programmers follow the design rules? Were the design rules helpful? What was good? What was bad?

- *Evidence for fault-tolerance* — Is the system fault-tolerant? The *raison d'être* for Erlang was to build fault-tolerant systems. Is there evidence that errors have occurred at run-time and been successfully corrected? Is the information produced when a programming error occurred good enough to subsequently correct the program?

- *Understanding the system* — Is the system understandable? Can it be maintained?

Rather than ask general questions about the properties of the system I am looking for specific evidence that the system behaves in a desirable manner. In particular:

1. Is there evidence to show that the system has crashed due to a programming error and that the error was corrected and that the system subsequently recovered from the error and behaved in a satisfactory manner after error correction had occurred?

2. Is there evidence to show that the system has run for an extended period of time and that software errors have occurred but that the system is still stable?

3. Is there evidence that code in the system has been updated "on the fly?"

4. Is there evidence that garbage collection etc works (ie that we have run a garbage-collected system for a long time without getting garbage collection errors?)

5. Is there evidence that the information in the error logs is sufficient for post-crash localization of the error?

6. Is there evidence that the code in the system can be structured in such a way that the majority of programmers do not need to be concerned with the details of the concurrency patterns used in the system?

7. Is there evidence that the supervision trees work as expected?

8. Is the code structured in a clean/dirty style?

Items 1, 2 and 5 above are included because we wish to test that our ideas about programming fault-tolerant systems work in practice.

Item 4 tests the hypothesis that long-term garbage collection can be used for real-time systems which have to operate over extended periods of time.

Item 6 is a measure of the abstraction power of the OTP behaviours. It is desirable for a number of reasons to "abstract out" details of concurrency for commonly recurring situations. The set of OTP behaviours is an attempt to do this. The extent to which programmers can to a first approximation ignore concurrency is an important measure of how suitable the OTP behaviours are for making system software. We can assess the extent to which concurrency can be ignored by observing how often programmers are forced to use explicit message passing and process manipulation primitives in their code.

Item 7 tests if the supervisor strategies work as expected.

Item 8 tests if it is possible to program according to the rules we gave in Appendix B. In particular the guidelines stress the importance of structuring the system in a clean/dirty manner. "Clean" code for our purposes is assumed to be side-effect free code, such code is much easier to understand than "dirty" code, ie code having side-effects.

Our entire system is concerned with the manipulation of hardware. This manipulation of hardware involves side-effects. Our concern therefore, is not whether side-effects can be avoided, but rather to what extent code with side-effects can be restricted to a small number of modules. Rather than having code with side-effects scattered in a uniform manner all over the system, it is desirable to have a small number of "dirty" modules with a large number of side-effects, combined with a much larger number of modules which are written in a side-effect free manner. An analysis of the code will reveal if such a structuring was possible.

Also of interest is "counter evidence." We would like to know about any cases where our paradigm breaks down, and if this breakdown was a major problem.

8.2 AXD301

The AXD301 [18] is a high-performance Asynchronous Transfer Mode (ATM) switch produced by Ericsson. The system is built from a number of scalable modules—each module provides 10 GBit/s of switching capacity and up to 16 modules can be connected together to form a 160 GBit/s

switch.

The AXD301 is designed for "carrier-class" non-stop operation [70]. The system has duplicated hardware which provides hardware redundancy and hardware can be added or removed from the system without interrupting services. The software has to be able to cope with hardware and software failures. Since the system is designed for non-stop operation it must be possible to change the software without disturbing traffic in the system.

8.3 Quantitative properties of the software

Here I report the result of an analysis of a snapshot of the AXD301 software. The snapshot represents the state of the system as it was on 5 December 2001.

The analysis is only concerned with the quantitative properties of the Erlang code in the system. The gross properties of the system are as follows:

Number of Erlang modules	2248
Clean modules	1472
Dirty modules	776
Number of lines of code	1136150
Total number of Erlang functions	57412
Total dumber of clean functions	53322
Total number of dirty functions	4090
Percentage of dirty functions/code lines	0.359%

In the above table the code has been subject to a simple analysis which superficially classifies each module or function as "clean" or "dirty." A module is considered dirty if any function in the module is dirty, otherwise it is clean. To simplify matters I say that a function is dirty if it sends or receives a message or if it calls one of the following Erlang BIFs:

```
apply, cancel_timer, check_process_code,
delete_module, demonitor, disconnect_node, erase,
group_leader, halt, link, load_module, monitor_node,
open_port, port_close, port_command, port_control,
process_flag, processes, purge_module, put, register,
registered, resume_process, send_nosuspend, spawn,
spawn_link, spawn_opt, suspend_process, system_flag,
trace, trace_info, trace_pattern, unlink, unregister,
yield.
```

The reason for this classification is that any code fragment which calls one of these BIFs is potentially dangerous.

Notice that I have chosen a particularly simple definition of "dirty." At first sight it might appear that it would be better to recursively define a module as being dirty if any function in the module calls a "dangerous" BIF or a dirty function in another module. Unfortunately with such a definition virtually every module in the system would be classified as dirty.

The reason for this is that if you compute the transitive closure of all functions calls exported from a particular module, the transitive closure will include virtually every module in the system. The reason why the transitive closure is so large is due to "leakage" which occurs from many of the modules in the Erlang libraries.

We take the simplifying view that all modules are well-written and tested, and that if they do contain side-effects, that the module has been written in such a way so that the side effects do not leak out from the module to adversely affect code which calls the module.

With this definition 65% of all modules are clean. Since any module is considered dirty if it contains a single dirty function, it is more interesting to look at the ratio of clean/dirty functions. A function will be considered dirty if a single call is made to unclean BIF. Viewed at a function level 92% of all functions appear to be written in a side-effect free manner.

Note that there were a total of 4090 dirty functions contained in 1.13 million lines of code, this is less than four dirty functions per thousand lines of code.

The distribution of dirty functions is shown in Figure 8.1. The distri-

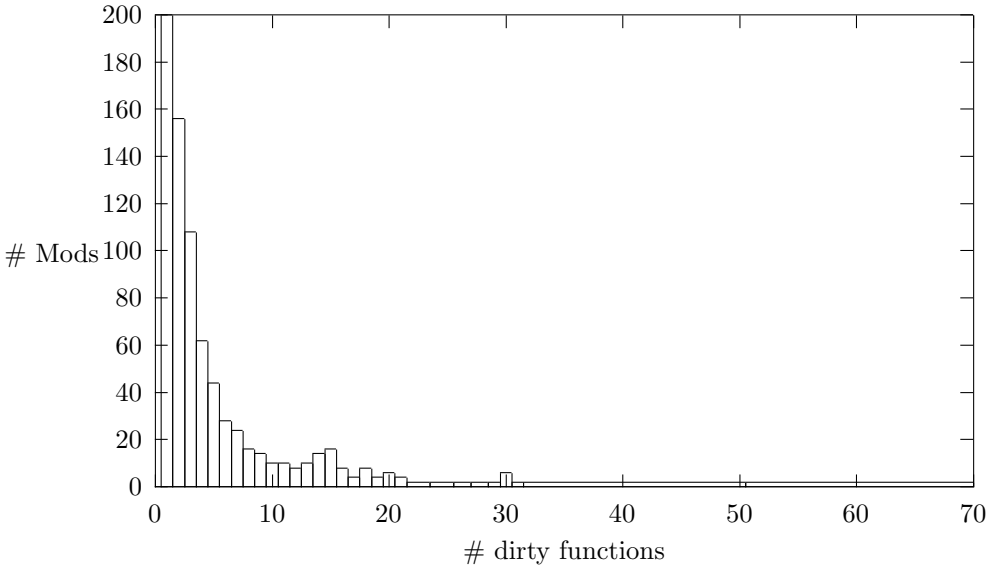


Figure 8.1: Distribution of dirty functions

bution of dirty functions is both encouraging and discouraging. The good news is that 95% of all dirty functions are found in slightly over 1% of the modules. The bad news is that there are a large number of modules with a very small number of impure functions. For example, there are 200 modules with 1 dirty function, 156 with 2 dirty functions etc.

The interesting thing about this data is that there has been no systematic effort to make the code "pure." The "emergent style" of programming seems therefore, to favour a style of programming where a small number of modules have a large number of side-effects, together with a larger number of modules having very few side-effects.

The Erlang programming rules favour this style of programming, the intention is to get the more experienced programmers to write and test the code that has lots of side-effects. Based on the AXD301 code it might be a good idea to explicitly define which modules are allowed to contain side-effects and enforce this with some kind of quality control.

If we look in detail at *which* primitives were called that could introduce side-effects, we get the following ordering:

`put` (1904), `apply` (1638), `send` (1496), `receive` (760), `erase` (336), `process_flag` (292), `spawn` (258), `unlink` (200), `register` (154),

spawn_link (126), link (106), unregister (38), open_port (20), demonitor (14), processes (14), yield (12), halt (10), registered (6), spawn_opt (4), port_command (4), trace (4), cancel_timer (2), monitor_node (2).

The most common primitive was put which was used 1904 times etc.

From this we can see that some of the Erlang primitives in our "black list" have never been used at all. The most common primitive which introduces a side-effect is put—depending upon how put is used this may or may not be dangerous. One common use of put is to assert a global property of a process which is used for debugging purposes. This is probably safe, though no automatic analysis program could infer this fact.

The dangerous side-effects are those which change the concurrency structure of the applications, thus modules which call the link, unlink, spawn or spawn_link, primitives must be carefully checked.

Even more dangerous is code that might evaluate halt or processes—I would assume that such code is very carefully checked.

8.3.1 System Structure

The AXD301 code is structured using the OTP supervision trees and the overall structure of the AXD301 code can be inferred primarily from the shape of these trees. Interior nodes in the supervision tree are themselves supervisor nodes, terminal nodes in the tree are OTP behaviours or more specialised application specific processes.

The AXD system supervision tree had 141 nodes and used 191 instances of the OTP behaviours. The number of instances of the behaviours were:

gen_server (122), gen_event (36), supervisor (20), gen_fsm (10), application (6).

gen_server and to a lesser extent gen_event dominate—there being 122 instances of the generic server. One interesting point to note is how few behaviours are required. The client-server abstraction (gen_server) is so useful that 63% of all generic objects in the system are instances of client-servers.

In the OTP libraries a supervisor starts an application by calling a function in the so-called `child_spec` of the processes which the supervisor controls. Among other things the "child specification" contains a `{Mod,Func,Args}` tuple which is used to specify how to start the supervised process.

This method of starting a supervised process is completely general, since any arbitrary function can be started by the supervisor. In the AXD301 case, the full generality of this method was not used, instead one of three different startup methods was used for all supervision hierarchies. Of these three methods, one method dominated, and was used for all except three of the supervision trees.

The AXD301 architects defined one master supervisor, which could be parameterised in a number of standardised ways. The AXD301 supervisors were packaged as conventional OTP applications, whose behaviour was described in so-called `.app` files [34]. Analysing all the AXD301 apps gives us a good overall idea of the static structure of the AXD software.

There were a total of 141 `.app` files the AXD software. These 141 files represent 11 disjoint supervision trees. Most of the supervision trees are very flat and do not have a complex structure.

A simple way to show this structure is to plot the trees in a simple ASCII display. Here, for example, is one of the eleven top-level trees, responsible for "Standby" services

```
|-- chStandby           Standby top application
|   |-- stbAts          Standby parts of ATS Subsystem
|   |   |-- aini_sb     Protocol termination of AINI, ...
|   |   |-- cc_sb       Call Control.Standby role
|   |   |-- iisp_sb     Protocol termination of IISP ...
|   |   |-- mdisp_sb    Message Dispatcher, standby role
|   |   |-- pch_sb      Permanent Connection Handling ...
|   |   |-- pnni_sb     Protocol termination of PNNI ...
|   |   |-- reh_sb      Standby application for REH
|   |   |-- saal_sb     SAAL, standby application.
|   |   |-- sbm_sb      Standby Manager, start standby role
|   |   |-- spvc_sb     Soft Permanent Connection ...
|   |   |-- uni_sb      Protocol termination of UNI, ...
|   |-- stbSws          Standby applications - SWS
|   |   |-- cecpSb      Circuit Emulation for AXD301, ...
```

```
|   |   |-- cnh_sb      Connection Handling on standby node
|   |   |-- tecSb       Tone & Echo for AXD301, SWS, ...
```

As can be seen the tree has a very simple structure, being flat rather than deep. There are two main sub-nodes in the tree, and the supervisor structure underneath the sub-nodes is flat.

Note that displaying the data in this manner only shows the organisation of the supervisor nodes themselves. The actual processes which were sub-nodes to the leafs in the tree are not shown, nor is the type of supervision (ie and or or supervision).

The reason why the trees are flat rather than deep reflects experience gained with the AXD software. Put simply *"Cascading restarts do not work."*

Wiger [69] who was the chief software architect of the AXD301 found that restarting a failed process with the same arguments often worked, but that if the simple restart procedure failed, then cascading restarts (ie restarting the level above) generally did not help.

Interestingly, in 1985, Gray had observed that most hardware faults were transient and could be corrected by reinitialising the state of the hardware and retrying the operation. He conjectured that this would also be true for software:

> *I conjecture that there is a similar phenomenon in software — most production faults are soft. If the program state is reinitialized and the failed operation retried, the operation will usually not fail the second time.* — [38]

The full generality of the model presented in this thesis, namely that of trying a simpler strategy in the event of failure was only partially exploited. This particular exploitation was accidental rather than by design—the OTP libraries themselves which provide interfaces to the file system and to system level services like sockets etc are written in such a manner as to protect the integrity of the system in the event of a failure. So, for example, files or sockets are automatically closed if the controlling processes for the file or socket terminates for any reason.

The level of protection provided by the OTP library services automatically provides the "simpler level of service" which is implied by our model for fault-tolerant computing.

8.3.2 Evidence for fault recovery

In the following sections I present evidence that the error-recovery mechanism have worked as planned. This evidence is based on an analysis of entries contained in the Ericsson trouble report database. When I have quoted from the data in the trouble report database I have not added anything to the entries, but I have removed irrelevant detail.

8.3.3 Trouble report HD90439

Trouble report HD90439 from 14 May 2003 has the following information:

```
1. Description

Heading: CRASH REPORT - Performance measurements on ET2 issue
Priority: C:   3 M, Minor fault or opinion: no traffic disturbance
Status: FI:   Finish
Hot TR: NO

...

2. Observation   Top of page

EFFECT:
CRASH REPORT - Performance measurements on ET2 issue

DESCRIPTION:
Node: AXD305 R7D PP6
Customer: ***************

=CRASH REPORT==== 14-May-2003::14:05:00 ===
  crasher:
    pid: <0.5605.0>
    registered_name: []
    error_info: {function_clause,[{etcpPrm,get_hwm_base,[ne_cv_1_ga,et2]},
                                  {prmMibExt,create_mep,3},
                                  {prmMibExt,get_counter_values,4},
                                  {perfCollect,collect_group,2},
                                  {proc_lib,init_p,5}]}
    initial_call: {perfCollect,collect_generic,
                               [{observed_object_group,
                                    {groupCb,prmMibExt},
                                    1504,
                                    [127],
                                    undefined},
                                63220104300]}
```

```
35      ancestors: [perfServer,perfSuper,omsSuper,omAxd301Super,<0.4396.0>]
36      messages: []
37      links: [<0.4523.0>]
38      dictionary: [{{eqm_mi_apply,{em,70},if_type_to_sublayer,1},
39                    {intfIfDbase,if_type_to_sublayer_int}}]
40      trap_exit: false
41      status: running
42      heap_size: 121393
43      stack_size: 23
44      reductions: 1332403
45    neighbours:
46
47  MEASURES:
48  Performance measurements were turned off and the crash report stopped
49  occurring
50
51  ...
52
53  4. Answer
54
55  ...
56  P R O B L E M & C O N S E Q U E N C E
57  ============================================================
58
59  A combination of the wildcard implementation together with a DS1
60  measurement can cause the described crash. The effect is that the
61  measurement fails.
62
63
64  S O L U T I O N
65  ============================================================
66
67  The fault has been detected, but it will not be released in R7D unless
68  it is important for a customer.
69
70  The wildcard implementation is greatly improved in R8B where this problem
71  does not exist. In R7D, we recommend to specify the PDH interfaces to be
72  included
73  in the DS1 measurement.
```

Crash number 90439 is fairly typical, it illustrates the situation where a hardware errors occurred, is corrected and the system reverted to normal use. The crash report is in lines 23–27 of the error log and it contains the following information:

```
{function_clause,
  [{etcpPrm,get_hwm_base,[ne_cv_1_ga,et2]},
   {prmMibExt,create_mep,3},
   {prmMibExt,get_counter_values,4},
   {perfCollect,collect_group,2},
   {proc_lib,init_p,5}]}
```

When `etcpPrm:get_hw_base(ne_cv_l_ga,et2)` is called the function call fails with a pattern matching error. Interestingly, this error occurred on 14 April 2003, the snapshot of the system that I had access to is from 5 December 2001, and therefore I reasoned that the error might have been present in my snapshot code. To satisfy my curiosity I checked the code in `etcpPrm`. The code looked like this:

```
get_hwm_base(locd_cnt, et155) ->
    ?et155dpPmFm_MEPID_Frh_LOCDEvt;

... 386 lines omitted ...

get_hwm_base(fe_pdh2_uat_tr, et2x155ce) ->
    ?et2x155cedpPmFm_MEPID_Frh_FE_E1_UAT_Tr.
```

Indeed there was no pattern which would match the calling arguments, so clearly this error would occur if called with the arguments shown in the error log. I was able to clearly locate the error and understand why the program has failed even though I didn't have the faintest idea what the error means.

It is also encouraging to note that the programmer who wrote this code had not programmed in a defensive manner, ie they had not written code like this:

```
get_hwm_base(locd_cnt, et155) ->
    ?et155dpPmFm_MEPID_Frh_LOCDEvt;
... 386 lines omitted ...
get_hwm_base(fe_pdh2_uat_tr, et2x155ce) ->
    ?et2x155cedpPmFm_MEPID_Frh_FE_E1_UAT_Tr;
get_hw_base(X, Y) ->
    exit(....).
```

Instead they had written their code exactly in the style which I recommended on page 108. Recall that the motivation for writing code in this manner is that a defensive style of programming is unnecessary since the Erlang compiler automatically adds additional information that is suitable for debugging the program.

8.3.4 Trouble report HD29758

Crash number HD29758 is more interesting. Reading the log and the subsequent comments of the engineers who studied the problem we can see that an error occurred at run-time which must have been corrected. Even though an error occurs it does not affect the traffic, and it is deemed not worthy of correction.

Here is a section extracted from the trouble report:

```
2   T R O U B L E  D E S C R I P T I O N

R7D NIT test case 3.4.1.2, takeover of OM process by CH CP.
Traffic continues to run successfully,
but we get several ERROR REPORTS in the
CP being blocked.

 Here's an example, see enclosure for erlang log:

  =ERROR REPORT==== 5-Apr-2002::09:03:55 ===
      error_info: {{case_clause,[]},
                   [{mdispGenServ,from_plc,4},
                    {mdispGenServ,handle_info,2},
                    {gen_server,handle_msg,6},
                    {proc_lib,init_p,5}]}
      msg_info_Tag: from_plc
      msg_info_MFA: {mdispGenServ,from_plc,
                                    [old_hc,
                                     {hcid,
                                         {ncs,mlgCmHcc,{97575,0}},
                                         msgQueue,
                                         undefined}]}
      msg_info_PlcResult: true
      node: 'axd301@cp1-1'
      proc_info: [{pid,{<0.20804.4>,"MDISP Server"}},
                  {message_queue_len,0},
                  {dictionary,[{mdispPerfPid,<0.20805.4>},
                               {'$ancestors',
... many lines omitted ...

4.2 Answer Text

```

```
33 The fault has been solved in version R8B of block
34 MDISP. No solution is planned in earlier versions
35 for mainly two reasons:
36
37 1) The fault has not yet given any obvious negative
38 effects on traffic handling.
39
40 ...
```

Lines 10–28 show that an error has occurred. Line 4 and lines 37–38 show that despite the error the system functions without any *"obvious negative effects on traffic handling."*

This error is rather nice, it shows that for a certain execution path an error is encountered. The system detects and recovers from the error, so for this particular error the system behaved in a reasonable manner in the presence of a software error.

8.3.5 Deficiencies in OTP structure

One interesting area in which the Erlang model of programming did not work was in the handling of call setup and call termination used in the AXD301 software.

In the section on the philosophy of COP, I argued for a 1:1 mapping of the problem structure onto the software architecture. In one important part of the AXD301 software this mapping was not possible. This part of the software had to do with the setup and termination of calls.

In order to understand this point I must go into a little detail about one of the most important services offered by the AXD301 switch. The AXD301 is a switching system, as such it is responsible for the maintenance of a large number of connections.

At any one time, one module in the system will be handling a large number of virtual channels. Each channel represents a single connection. Typically, a single node might handle up to 50,000 connections. Connections are in one of three possible states:

1. *Setup* — In this state a new connection being established. There is intensive signalling.

2. *Connected* — In this state the connection is established. There is very little signalling, only monitoring of the on-going connection.

3. *Terminating* — In this state the connection is terminating. There is some signalling between the end-points of the connection.

The setup and termination phases of a connection are very quick, and typically take a few milliseconds. Termination is somewhat simpler than setup. In the connection phase only monitoring is involved, the connection phase is typically many orders of magnitude longer than the setup and termination phases. Connections could range from seconds to hours, or even years.

At any time there might be up to 50,000 connections per node, the vast majority of which will be in the connected state. The system is dimensioned for a maximum of 120 calls/second—this number refers to the number of simultaneous calls which can be in the setup or termination phase, not to the number of calls which are in the connected phase.

Modelling the natural concurrency of the problem needs about six concurrent processes per call setup/termination. Having six processes per connection during setup and (say) a couple of processes per connected call requires a few hundred thousand Erlang processes. Most of these processes do nothing and are suspended for a very long time (these are the processes which monitor the connected calls)—this could not be implemented for a number of reasons.

Firstly, processes take space, even if they are not doing anything. Secondly, any state involved in the connected phase must be replicated across a physical hardware boundary. This is in order to provide for continuous service in the presence of a hardware failure.

The AXD301 tries to minimise the number of active processes used for connection handling and uses the following strategy.

1. During call setup six processes are created to handle each call.

2. If the call setup succeeds, then all relevant information about the call is reduced to a “call record” which describes the call. The six processes used in call setup are destroyed and the call record which

describes the call is stored locally. Finally, an asynchronous message containing a copy of the call record is sent to the backup node for the current node.

3. At call termination, the process structure required to terminate the call is created and initialized from the data in the call record. Call termination processing occurs and all the processes involved are destroyed.

Because the call setup process is well understood, the memory requirements for a process in the setup phase is well understood. The maximum stack and heap size needed by the setup processes is well known and has been established by a number of measurements. Using this data it is possible to initialise the six processes needed for call setup with sufficiently large initial stacks and heaps so that no garbage collection occurs during call setup.

During setup no attempt is made to replicate the state of the calls across a physical boundary, so if the system crashes, the call will be lost. In this event the client will detect the failure and merely try again. In the event of hardware failure the retry will be directed to a new hardware module and should succeed. Since call setup is so quick, this is hardly a problem.

When the setup phase is complete, all information is reduced to a "call record" (this is about 1 KB/call) and the six processes used in call setup are destroyed. An asynchronous message containing the call record is sent to the backup processor. Hardware units are always configured in pairs. For the pair (`A`, `B`), the machine `A` is considered a backup machine for machine `B` and machine `B` is a backup for machine `A`.

Signals from a particular client are arbitrated by a hardware dispatcher unit, which sends all signals for a particular call to the "master" unit for the call. If the master unit fails and the call is in the *connected* state then the arbitrator will direct signals to the backup unit.

Call termination, or modification to the call is the inverse of call setup. When the system detects that some operation is to be performed on the call, the call record is retrieved and the process structure necessary to manipulate the call is created and initialized with data from the call record. Thereafter, call processing proceeds as for call setup.

This model minimizes the number of processes to the active set of processes needed to perform a particular operation. When the application reaches a point in time where there is little active processing of the data, the processes are destroyed and the relevant data needed to re-create the process structure is stored in a database and is asynchronously replicated on a backup machine.

This approach has the advantages of having a flexible process structure which is needed in the complex phases of call setup or termination—but when the processes are inactive, storage is reclaimed by deleting the processes and storing their state data in stable storage.

The fact that the state information is asynchronously replicated increases throughput in the system without compromising the integrity of the system.

This solution fits nicely with the Erlang processes model. Since processes are light-weight, we can create and destroy them "on-demand." Systems with heavy-weight processes could use a similar solution, where instead of destroying a process when it has finished, the process is recycled, and a pool of processes is maintained by the system.

At the time when the AXD301 software was developed there was an upper limit on the total number of Erlang processes which could run at the same time in the system. This number, while large, was not large enough to allow hundreds of thousands of processes in the system. Later versions of the system allow hundreds of thousands of Erlang processes, though not millions of processes.

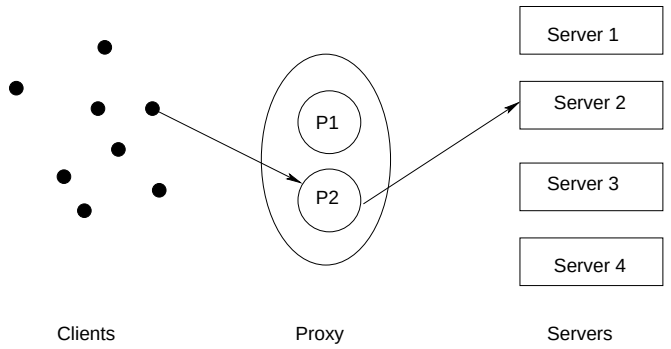
Even though the designer of a system should not, at a first approximation have to worry about the total number of processes in the system, they should be able to crudely dimension the system and have a rough idea as to how many processes will be needed throughout the operation of an application.

For very long-lived processes, storing the process state in a data base when the process is not active, is an attractive alternative. This method has the added advantage that by replicating the state on stable storage the same software can be made fault-tolerant.

8.4 Smaller products

The second case study concerns two products developed by Bluetail AB. The first of these products was the "Bluetail Mail Robustifier" (BMR) which is described in [11].

8.4.1 Bluetail Mail Robustifier



The Bluetail Mail Robustifier was a product designed to increase the reliability of existing e-mail services. The BMR was designed as a proxy which was placed between the clients which wished to make use of some e-mail services, and a number of e-mail servers.

As far as the clients were concerned all the servers had the same IP address (the address of the proxy)—internally the proxy had at least two physical machines (for fault-tolerance) which could intercept and relay messages to the back-end servers. The back-end servers were themselves machines running standardised mail servers. The BMR concentrated on three mail protocols SMTP [57], POP3 [52] and IMAP4 [27].

The BMR was shipped to its customers as a "turn-key" system. The requirements [11] for the BMR were given as:

1. *Down times should be a few minutes per decade.*
2. *If a mail server fails some other server should take over with a minimum of delay, clients should not notice that the server has failed.*

3. *It should be possible to remotely manage the system. We would like to add or remove mail servers or take them out of service without interrupting the quality of service.*
4. *It should be possible to upgrade the BMR software itself without stopping the system.*
5. *In the event of an error in the BMR software it should be possible to revert to a previous version of the software without stopping the system.*
6. *The system must work together with and improve the performance of existing mail systems.*
7. *The system should work on existing hardware and run on existing operating systems.*
8. *The system should be at least as efficient as a conventional imperative language implementation.*
9. *The system should have a conventional Unix command line interface, and conventional Unix manual pages.*
10. *The system should have a GUI interface.*

Points 1, 3 and 4 are typical requirements for a soft real-time system, in particular point 4 is a typical requirement for this kind of system but is rarely found in the requirements for a system that is only to be used for a short period of time. Point 5 is related to point 4—as far as possible we want the operation of the software to be entirely autonomous and to require minimal operator intervention. The fact that the BMR could be remotely managed and that upgrades and downgrades to the system could be handled automatically and remotely was an important factor in selling the product. Indeed, one of the main reasons for buying the BMR was to relieve the burden of manually monitoring and maintaining an e-mail system.

The BMR was written using 108 Erlang modules and had 36,285 lines of Erlang code. It was written from scratch and delivered to the first customer within six months of the project start. The BMR has been in live

operation with the Swedish Telenordia ISP since 1999 and handles millions of emails per year.

BMR implemented its own release management system, as an extension to the release behaviour in the OTP system. The BMR system is an intrinsically distributed system. It is desirable that software upgrade in a distributed system has "transaction" semantics, that is, either the software upgrade works in its entirety on all nodes of the system, or, that it fails and no software is changed on any node.

In BMR two versions of the entire system could co-exist, a old version and a new version. Adding a new version of the software, makes the current version of the software the old version, and the added version becomes the new version.

To achieve this, all BMR software upgrade packages were written in a reversible manner, ie it should be possible to not only perform a dynamic upgrade from an old version of the software to a new version, but also to convert from a new version back to the old version.

Upgrading the software on all nodes was done in four phases.

1. In phase one the new software release is distributed to all nodes—this always succeeds.

2. In phase two the software on all nodes is changed from the old version to the new version. If the conversion on any node fails, then all nodes running new versions of the software are backed to run the old version of the software.

3. In phase three all nodes in the system run the new software, but should any error occur, then all nodes are backed to run the old software. The system has not yet committed to run the new software.

4. After the new system has run successfully for a sufficient time period the operator can "commit" the software change. Committing the system (phase four) changes the behaviour of the system. If an error occurs after a commit then the system will restart using the new software and not revert to the old software.

Interestingly, almost exactly the same mechanism is used in the X2000 system [63] developed by NASA, for their deep-space applications, where software applications also have to be upgraded without stopping the system.

In addition the BMR upgrade system had to allow for the possibility that a node in the distributed system was "out-of-service" at the time when the software upgrade was being performed. In this case, when the node was re-introduced to the system it would learn about any changes that had been made to the system during the time it was unavailable, and any necessary software upgrades would be performed.

8.4.2 Alteon SSL accelerator

The Alteon SSL[3] Accelerator was the first product to be produced after Bluetail AB was acquired by Alteon Web Systems. An SSL accelerator is a hardware appliance containing special purpose hardware for speeding up cryptographic computations. The SSL accelerator is marketed and sold by Nortel Networks. The control system for the SSL accelerator is written in Erlang.

> *According to Infonetics Research, the Alteon SSL Accelerator is the leading SSL Accelerator appliance in the market. With more SSL Accelerators deployed than any other vendor, Nortel Networks leads the market with innovative new applications and features such as back-end encryption, integrated load balancing, session persistence, application address translation, Layer 7 filtering, and secure global server load balancing (GSLB). After winning all evaluation categories, Network Computing named Nortel Networks Alteon SSL Accelerator "King of the Hill" in their latest SSL Accelerator bake-off, citing industry-leading performance, features, and manageability as distinguishing attributes.* — [53]

[3] Secure Socket Layer.

The SSL accelerator is produced as a hardware appliance. Interestingly it was produced in a very short time (less than one year) and rapidly became market leader having won the "best in test" awards in all categories awarded by "Network Computing."

The software architecture for the SSL accelerator was derived from the generic architecture used in the Bluetail Mail Robustifier.

8.4.3 Quantitative properties of the code

Unfortunately, Nortel Networks would not let me analyse their source code in any detail, so I can only report on the grossed up properties of their code. This data is derived for all Bluetail/Alteon Erlang products. It does not distinguish the individual products:

Number of Erlang modules	253
Clean modules	122
Dirty modules	131
Number of lines of code	74440
Total number of Erlang functions	6876
Total number of clean functions	6266
Total number of dirty functions	610
Percentage of dirty functions/code lines	0.82%

The products made extensive use of the OTP behaviours using 103 behaviours in all. Again `gen_server` dominated. The number of times the different behaviours were used were:

`gen_server` (56), `supervisor` (19), `application` (15), `gen_event` (9), `rpc_server` (2), `gen_fsm` (1), `supervisor_bridge` (1).

There is not so much to say about these figures, apart from the fact that at first sight the AXD project used relatively longer functions and relatively fewer behaviours than the Bluetail/Alteon projects.

I interviewed the people who had programmed the SSL accelerator. They told me that in practice the architecture worked well and that failures due to software unrecoverable software errors had not occurred, but that they kept no records which could confirm this observation.

They also said that the product upgrade mechanism had evolved beyond that developed in the OTP system. Any product release increment had to be designed in such a way that it was totally reversible. That is, when planning to go from version `N` of the system to version `N+1` they would not only have to write code to go from version `N` to version `N+1` but also they would have to write code to go from version `N+1` to version `N`.

This discipline was strictly followed, it was therefore possible to "roll-back" the system from its current incarnation to the very first release. For commercial reasons, Nortel did not wish to release any detailed information about the details of this process.

8.5 Discussion

Before I started these case studies I had a fairly clear idea as to what parameters it would be possible to measure. The ultimate test of a technology is ask the users of the technology the following question:

"Does it work?"

If you ask this question to the people at Ericsson or Nortel they shake their heads in amazement and say:

"Of course it works!"

I had hoped that they would be able to provide me with clearly documented evidence that the system does indeed work in the manner that I have described in this thesis. I would liked to have found evidence in the log files that the code upgrades had worked, that the system had crashed and recovered, that the system had run for thousands of hours without interruption etc.

The people who I interviewed were happy to tell me that this was the case, that code-upgrades etc did work as expected. But they could not produce any collaborative evidence that this was the case (ie there were no records in the log files that code upgrades etc had occurred).

The reason that there was no documentary evidence was that the system had not been instrumented in such a way as to report to the log files that things like code upgrade had occurred, and thus there was no lasting evidence. Interestingly there was no counter-evidence to suggest that the

procedures had failed. I assume that if the procedures had failed then there would have been a lot of trouble reports etc which documented this fact.

Evidence for the long-term operational stability of the system had also not been collected in any systematic way. For the Ericsson AXD301 the only information on the long-term stability of the system came from a power-point presentation showing some figures claiming that a major customer had run an 11 node system with a 99.9999999% reliability, though how these figure had been obtained was not documented.

In the case of the BMR and the SSL accelerator there was no hard evidence that the system had run reliably for extended periods of time. The reason for this is that such information is not recorded in the log files.

The BMR is, of course, a fault-tolerant distributed system, having several interconnected nodes. In such a system the failure of an individual node is not a "noteworthy event," since the system is designed from the beginning to survive the crash of a single node.

If single nodes do crash (which they obviously do, since there is anecdotal evidence that this is the case) then the performance of the system as a whole is slightly degraded, but not to the point where it becomes an operational problem.

I am unaware of any entire system failures (where all nodes fail)—were this to be the case I would suspect that this is due to a massive hardware failure affecting all machines in the system. Such failures appear to be extremely rare, and even if they did occur the cause of failure has nothing to do with the software, but is an entirely different hardware issue.

In a distributed system the perception of failure, and even our language for talking about failure need modification. It hardly makes sense to talk about total system failure (since this is such an uncommon event)—instead we need to talk about some measure of degradation of service quality.

The software systems in the case study are so reliable that the people operating these system are inclined to think that they are error-free. This is not the case, indeed software errors did occur at run-time but these errors were quickly corrected so that nobody ever noticed that the errors had occurred. To get accurate statistics on the long-term stability one would have to record all start and stopping times of the system and measure a

number of parameters that indicated the "health" of the system. Since the system appears to be entirely "healthy" such statistics never seem to have been collected in any systematic way.

As regards an analysis of the AXD301 code base I had hoped to find evidence that the programming rules were being used correctly—I would liked, for example, to have seen a clear separation into "clean" and "dirty" code. I accept that certain parts of the system will always be written in a suspect manner (for reasons of efficiency, or for some other reason) but I would have liked this code to be separated in a clear manner from the main body of the code, and for more stringent test procedures to be applied to this code etc.

This was not the case. A majority of the code was clean, but the distribution of dirty code was not a pure "step" function (ie there was no clear division where I could say, "this code is bad, stare at it carefully" and "this code is good") but rather a spread out distribution where there were a small number of modules which had a lot of side-effects (this doesn't worry me), but more troublesome a largish number of modules with only one or two calls to dirty primitives.

Without a deeper understanding of the code than is possible here it is impossible to say if this is in the nature of the problem, and that these calls with potential side-effects introduce problems into the system, or if they are harmless.

In any case, the message is clear. Programming rules alone are insufficient to cause the system to be written in a particular way. If one wished to enforce the partitioning of code into clean and dirty code then tool support would have to be added and the policies would have to be enforced in some manner. Whether or not this is desirable is debatable—probably the best approach is to allow the rules to be broken and hope that the programmers know what they are doing when they break a programming rule.

The ultimate test of a technology is, of course, the "did-it-work test." In the case of the AXD301 and the Nortel SSL accelerator the answer was a resounding "yes." Both these products are small niche products, but both of them are market leaders in their niches.

9 APIs and Protocols

When we have built a software module we need to describe how it is to be used. One such method is to define a programming language API for all the exported functions that are in the module. To do this in Erlang we could use the type system outlined on page 80.

This method of defining API's is widespread. The details of the type notation vary from language to language. The degree to which the type system is enforced by the underlying language implementation varies from system to system. If the type system is strictly enforced then the language is called "strongly typed," otherwise it is called "untyped"—this point often causes some confusion, since many languages which need type declarations have type systems which can easily be broken. Languages like Erlang, need no type declarations, but are "type safe," meaning that the underlying system cannot be broken in such a way as to damage the system.

Even if our language is not strongly typed, the existence of type declarations provides valuable documentation, and can be used as input to a dynamic type checker which can be used for run-time type checking.

Unfortunately, API's written in the conventional manner are not sufficient to understand the operation of a program. For example, consider the following code fragment:

```
silly() ->
    {ok, H} = file:open("foo.dat", read),
    file:close(H),
    file:read_line(H).
```

According to the type system and the API given in the example on page 80 this is a perfectly legal program and yet it is obviously total nonsense, since we cannot expect to read from a file once it has been closed.

To remedy the above problem we could add an additional state parameter. In a fairly obvious notation, the API could be written something like this:

```
+type start x file:open(fileName(), read | write) ->
           {ok, fileHandle()} x ready
         | {error, string()}  x stop.

+type ready x file:read_line(fileHandle()) ->
           {ok, string()} x ready
         | eof x atEof.

+type atEof | ready x file:close(fileHandle()) ->
           true x stop.

+type atEof | ready x file:rewind(fileHandle()) ->
           true x ready
```

This model of the API uses four state variables `start`, `ready`, `atEof` and `stop`. The state `start` means the file has not been opened. The state `ready` means the file is ready to be read, `atEof` means that the file is positioned at end-of-file. The server always starts in the state `start` and stops in the state `stop`.

The API now says that, for example, when we are in a state `ready`, that a `file:read_line` function call is legal. This will either return a string, in which case we stay in the state `ready` or it will return `eof` and we will be in the state `atEof`.

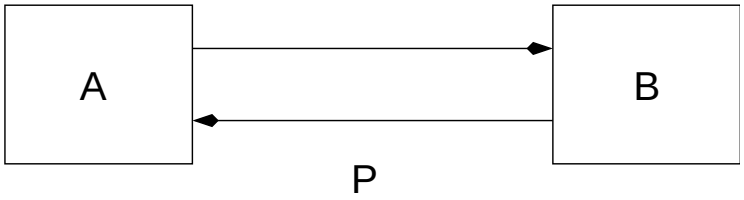
In state `atEof` we can close or rewind the file, all other operations are illegal. If we choose to rewind the file, the state will change back to `ready` in which case a `read_line` operation becomes legal again.

Augmenting the API with state information provides us with a method for determining if the sequence of allowed operation is in accordance with the design of a module.

9.1 Protocols

Having seen how we can specify sequencing in an API, there is an equivalent idea that is applicable to protocols.

Consider two components which communicate by pure message passing, at some level of abstraction we need to be able to *specify* the protocol of the messages which can flow between the components.



The protocol `P` between two components `A` and `B` can be described in terms of a non-deterministic finite state machine.

Assume that process `B` is a file server, and that `A` is a client program which makes use of the file server, assume further that sessions are connection oriented. The protocol that the file server obeys can be specified as follows:

```
+state start x {open, fileName(), read | write} ->
            {ok, fileHandle()} x ready
          | {error, string()}  x stop.

+state ready x {read_line, fileHandle()} ->
            {ok, string()} x ready
          | eof x atEof.

+state ready | atEof x {close, fileHandle()}->
            true x stop.

+state ready | atEof x {rewind, fileHandle()) ->
            true x ready
```

This specification says that if the file server is in the state `start` then it can receive a message of type `{open, fileName(), read|write}`, it

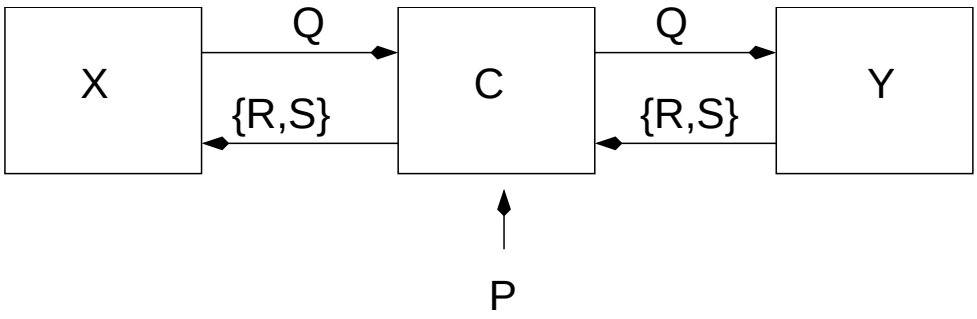


Figure 9.1: Two processes and a protocol checker

will then respond by sending out a message of type `{ok, fileHandle()}` and move to the state `ready` or it will respond by replying with the message `{error, string()}` and move into the state `stop`.

Given a protocol which is specified in a manner similar to the above it is possible to write a simple "protocol checking" program which can be placed between any pair of processes. Figure 9.1 shows a protocol checker `C` placed between two processes `X` and `Y`.

When `X` sends a message `Q` (a query) to `Y`, `Y` responds with a response `R` and with a new state `S`. The pair `{R,S}` can be type-checked against the rules in the protocol specification. The protocol checker `C` sits between `X` and `Y` and checks that all the messages between `X` and `Y` are according to the protocol specification.

In order to check the protocol rules the checker needs to have access to the state of the server, this is because the protocol specification might contain productions like:

```
+state Sn x T1 -> T2 x S2 | T2 x S3
```

In which case the observation of a reply message of type `T2` is not sufficient to distinguish between the output states `S2` and `S3`.

If we recall the simple generic server, shown on page 89, the main loop of the program was:

```
loop(State, Fun) ->
  receive
```

```
        {ReplyTo, ReplyAs, Q} ->
            {Reply, State1} = F(State, Q),
            Reply ! {ReplyAs, Reply},
            loop(State1, Fun)
  end.
```

Which can easily be changed to:

```
loop(State, S, Fun) ->
  receive
        {ReplyTo, ReplyAs, Q} ->
            {Reply, State1, S1} = F(State, S, Q),
            Reply ! {ReplyAs, S1, Reply},
            loop(State1, S1, Fun)
  end.
```

Where `S` and `S1` represent the state variable which was specified in the protocol specification. Note that the state of the interface (ie the value of the state variable used in the protocol specification) is not the same as the state of the server `State`.

Given such a change, the generic server becomes re-cast in a form which allows a dynamic protocol checker to be inserted between the client and the server.

9.2 APIs or protocols?

Up to now we have shown what are essentially two equivalent ways of doing the same thing. We can impose a type system on our programming language, or we can impose a contract checking mechanism between any two components in a message passing system. Of these two methods I prefer the use of a contract checker.

The first reason for this has to do with how we structure systems. In our model of programming we assume isolated components and pure message passing. The components themselves are considered "black boxes." From outside the black box *how* a computation is performed *inside* the black

box is totally irrelevant. The *only* thing which is important is whether or not the black box behaves according to its protocol specification.

Inside the black box, it may be desirable, for reasons of efficiency or otherwise to program using obscure programming methods and to break all rules of common sense and good programming practice. This does not matter in the slightest, provided the external behaviour of the system is consistent with the protocol specification.

By simple extension the protocol specification can be extended to specify the non-functional properties of a system. For example, we might add to our protocol specification language a notion of time, then we could say things like:

```
+type Si x {operation1, Arg1} ->
      value1() x Sj within T1
    | value2() x Sk after T2
```

Meaning that `operation1` should return a `value1()` type data structure within time `T1` or return something of type `value2()` after time `T2` etc.

The second reason has to do with *where* we do things in the system. Placing the contract checker *outside* a component in no way interferes with the construction of the component itself, moreover it allows a flexible way of adding or removing introspective testing powers to the system, which can be checked at run-time and which can be configured in a number of different ways.

9.3 Communicating components

How should Erlang talk to the outside world? — this question becomes interesting if we want to build distributed applications where Erlang is one of a number of communicating components. In [35] we can read that:

> *The two fundamental building blocks underlying any PLITS system are modules and messages. A module is a self-contained*

> *entity, something like a Simula or Smalltalk class, a SAIL process or a CLU module. It is not important for the moment which programming language is used to encode the body of the module; we wish to explicitly account for the case in which the various modules are coded in different languages on a variety of machines* – [35]

In order to make a system of communication components we have to agree on a number of different things. We need:

- A transport format and a way of mapping language entities into entities in the transport format.
- A system of types, built on top of the entities in the transport format.
- A protocol description language in terms of the system of types.

A method for doing this involving a transport format called UBF (short for Universal Binary Format), which was designed for rapid parsing was presented in [13]. A slightly revised version of the paper can be found in appendix C.

9.4 Discussion

I want to return to the central question of the thesis–How can we make a reliable system in the presence of errors? Stepping outside the system and viewing the system as a set of communicating black boxes is very helpful here.

If we formally describe the protocol to be obeyed on a communication channel between two black boxes then we can use this as a means for detecting and identifying errors, we can also say precisely which component has failed.

Such an architecture satisfies requirements R1–R4 on page 27–and therefore, by my reasoning can be used for programming error-resilient systems.

The "try something simpler" idiom (page 116) also applies. If a black-box implementation of a function fails to work, then we could revert to a simpler implementation, also implemented as a black box. The protocol checking mechanism could be used for making meta-level decisions as to which implementation should be used, choosing a simpler implementation if errors are observed. When the components reside on physically separated machines, the property of strong isolation is almost automatically enforced.

10 Conclusions

Theses are never completed—at some point, however, you just give up... In this thesis I have presented a number of things—a new programming language and a method for programming fault-tolerant systems. Erlang appears to be a useful tool for programming reliable systems—amazingly it is being used for a large number of applications which are far removed from the the problem domain that Erlang was designed to be used for.

10.1 What has been achieved so far?

The work described in this thesis, and related work performed elsewhere has demonstrated a number of different things, namely:

- That Erlang and the associated technology *works*. This is, in itself, an interesting result. Many people have argued that languages like Erlang cannot be used for full-scale industrial software development. Work on the AXD301 and on the Nortel range of products shows that Erlang is an appropriate language for these types of product development. The fact that not only are these products successful, but also that they are market leaders in their respective niches is also significant.

- That programming with light-weight processes and no shared memory works in practice and can be used to produce complex large-scale industrial software.

- That it is possible to construct systems that behave in a reasonable manner in the presence of software errors.

10.2 Ideas for future work

It would be a shame if the Erlang story had no future chapters, and there are a number of directions in which I would like to see the language develop.

- *Conceptual integrity* — can we develop Erlang so as to reinforce the *everything is a process* view of the world? Can we make the system, and Erlang code more regular and easier to understand?

- *Kernel improvements* — the statement on page 83 is a modified truth. The property of strong isolation is not strictly adhered to in any known implementation of Erlang. One process could affect another process in the system by allocating vast amounts of memory or by sending large numbers of messages to another process. A malicious process could destroy the entire system by creating large numbers of atoms and overflow the atom table etc. Current implementations of Erlang are not designed to protect the system from malevolent attacks. Kernel improvements are possible which would guard against many such attacks.

- *How can we program components* — the idea of a set of communicating processes leads naturally to the idea of writing the components in different languages. How can we do this?

10.2.1 Conceptual integrity

How can we make the system easier to understand? The Erlang programming model is *"Everything is a process."* We can emphasise this view by making a few minor changes to the languages. With these changes, most (if not all) BIFs become unnecessary. Let me assume the following:

- Everything is a process.

- All processes have names.
- Message passing is explicit.

I also introduce a new infix remote procedure call operator, called "bang bang" and written:

```
A !! B
```

If A is a Pid then this is short for:

```
A ! {self(), B},
receive
   {A, Reply} ->
        Reply
end
```

In fact A can be a Pid, an atom, a string or a list of Pid's atoms or strings.

[A1,A2,A3,..]!!X returns [V1,V2,V3,...] where V1 = A1!!X and V2 = A2!!X, etc. All the RPCs are performed in parallel.

10.2.2 Files and bang bang

Recall that I said earlier that everything is a process, and that all processes have names, thus files are processes.

If F is a file, then F !! read reads the file, and F !! {write, Bin} writes the file.

The following code fragment reads a file and copies it to a new location:

```
{ok, Bin} = "/home/joe/abc" !! read,
"/home/joe/def" !! {write, Bin}
```

This example reads three files in parallel:

```
[A, B, C] =
        ["/home/joe/foo",
        "http://www.sics.se/~joe/another_file.html",
        "ftp://www.some.where/pub/joe/a_file"] !! read
```

With some fairly obvious syntactic sugar.

Now suppose I am working at home, and keep a copy of my work on a remote host. The following code compares my local copy of the file with the copy on a backup machine, and if they are different updates the backup copy:

```
L = ["/home/joe/foo",
     Remote= "ftp://www.sics.se/~joe/backup/foo"],
case L !! read of
    {X, X}       -> true;
    {{ok,Bin},_} -> Remote !! {write, Bin};
    _ -> error
end
```

10.2.3 Distribution and bang bang

If a process name is a string of the form `"erl://Node/Name"` then the operation will be performed on a remote node instead of on the local node, that is:

```
"erl://Node/Name" !! X
```

means evaluate `Name !! X` on the node `Node` and return the result to the user. Thus:

```
{ok,Bin} =
  "erl://joe@enfield.sics.se/home/joe/abc" !! read,
"/home/joe/def" !! {write, Bin}
```

reads a remote file and stores it on the local computer.

10.2.4 Spawning and bang bang

In the beginning of the universe several "magic" processes are assumed to exist. These have names like "/dev/..."—these processes are pre-defined and do magic things.

In particular, "/dev/spawn" is a process spawning device. If you send "/dev/spawn" an Erlang fun it will send you back an Erlang Pid, so:

```
Pid1 = "/dev/spawn" !! fun() -> looper() end),
Pid2 = "erl://joe@bingbang.here.org/dev/spawn"
       !! fun() -> ... end
```

Creates two Erlang processes, one on bingbang.here.org. and the other on the local node.

Now we can throw away the Erlang primitive spawn. It is replaced by the spawn device.

10.2.5 Naming of processes

I talked about "/dev/spawn", here are a few more devices in the same flavour:

/dev/spawn	process spawner
/dev/stdout	stdout
/dev/log/spool/errors	error logger
/dev/code	code server
/proc/Name	processes
/Path/To/File	files
/dev/dets	dets tables
/dev/ets	ets tables
/dev/mnesia	mnesia tables
http://Host/File	read only files
erl://Node@Hist/..	remote processes

We also need to extend the process registration primitive, to allow us to register a string as a name for a process.

```
-module(vshlr4).

-export([start/0, stop/0, handle_rpc/2, handle_cast/2]).
-import(server1, [start/3, stop/1, rpc/2]).
-import(dict,    [new/0, store/3, find/2]).

start() -> start(vshlr4, fun handle_event/2, new()).

stop() -> stop(vshlr4).

handle_cast({i_am_at, Who, Where}, Dict) ->
    store(Who, Where, Dict).

handle_rpc({find, Who}, Dict) ->
    {find(Who, Dict), Dict}.
```

Figure 10.1: A Very Simple Home Location Register (revisited)

10.2.6 Programming with bang bang

I start with a simple example. Figure 10.1 is my old friend the home location register, revisited yet again. This time `vshlr4` is written as a pure Erlang function, and no attempt is made to hide the access functions inside stub function. Instead we *expose* the interface.

With the bang bang notation, the shell dialog shown on page 90 becomes:

```
1> vshlr4:start().
true
2> vshlr4 !! {find, "joe"}.
error
3> vshlr4 ! {i_am_at, "joe", "sics"}.
```

```
ack
4> vshlr4 !! {find, "joe"}.
{ok,"sics"}
```

We achieve the same results as before, but this time we have not encapsulated the remote procedure call in a stub function. Instead we have exposed it to the programmer.

10.3 Exposing the interface - discussion

Should we expose the interfaces or not? Let's imagine two ways of doing things. Suppose we have a remote file server, accessible through a variable `F1`, suppose also that the protocol between the file server and the client code is hidden in some module `file_server.erl`. Inside `file_server.erl` we find code like this:

```
-module(file_server).
-export([get_file/2]).
...
get_file(Fs, FileName) ->
    Fs ! {self(), {get_file, Name}},
    receive
        {Fs, Reply} ->
            Reply
    end
```

The access function `get_file/2` is used by client code to fetch a file from the file server. The statement:

```
File = file_server:get_file(F1, "/home/joe/foo")
```

is used to fetch a file. The code which does this is nicely hidden inside `file_server` and the user has no idea or need to know about the underlying protocol. In *Concurrent Programming in Erlang* we argued that hiding the details of an interface was good programming practice:

> *The purpose of interface functions is to create abstractions which hide specific details of the protocol used between the clients and the server. A user of a service does not need to know the details of the protocols used to implement the service, or the internal data structures and algorithms used in the server. An implementor of the service is then free to change any of these internal details at any time while maintaining the same user interface.*—[5] (pages 81–82)

An unfortunate consequence of this encapsulation, is that we cannot easily parallelise simultaneous calls to different file servers, nor can we stream requests to a single file server.

If we wish to perform several RPCs in parallel it might be better to dispatch all the requests, and then gather all the responses, but this cannot be done in all situations unless the interface is exposed.

10.4 Programming communicating components

Interestingly the dominant way of building Internet applications involves the use of:

- Isolated components.
- Pure message passing.
- Informal protocols.

In the Internet case the components really are physically isolated, for example, a client in Sweden might make use of the services of a server in Australia. Pure message passing with no shared data is the norm. This is possibly because just about the only way that the system builder can understand and build distributed applications involves the use of asynchronous

protocols described informally by the use of RFCs.[1] The problem with RFCs is that virtually every RFC defines its own ad hoc syntax for data transit, and that the allowed sequences of messages are often never described formally, but must be guessed by the implementor.

The use of a standard syntax for describing RFCs (for example, XML, lisp S-expressions or UBF terms) would be a great step forward, enabling a single parser to be used for all applications. The use of a contract checker as proposed in appendix C would also make programming Internet applications significantly easier, and hopefully would make the applications much more reliable.

Interestingly, the Internet model works well even without notification for the reasons for failure. If we have a pathological distrust of the component we are talking to, then this might be sensible, but for normal applications notification of failure reason would make implementing and debugging applications a lot easier.

The use of the contract checker also accurately pinpoints exactly where errors have occurred, this becomes increasingly important as more communicating components are added to the system.

While the model of communicating components is widely used for distributed applications, it is infrequently used for structuring single-node applications. In the interest of efficiency, designers reject the idea of using processes as protection domains for their software, and favour shared object models.

I believe that the model for programming single-node applications should be exactly the same as that used for programming distributed applications. Different components should be protected from each other by processes. It should be impossible for the components to damage each other and the components should communicate using defined protocols which are specified by a contract and enforced by a contract checker.

This way we can build reliable applications.

[1] Requests For Comments–a set of memos describing various aspects of the Internet, including all the major application protocols.

Acknowledgments

The Erlang system, the OTP libraries, and all the applications that have been built in Erlang are the result of a massive amount of work done by a large number of people.

It is always difficult to write the acknowledgments section, since I don't want to miss out anybody who has contributed to the system, and I don't want to make any errors—I shall try to be as accurate as possible, for any omissions or inaccuracies I apologise in advance.

I would like to start by thanking all the "management types" who made this possible in the first place. Bjarne Däcker who was my boss at Ericsson has alway been enthusiastic about our work, and fought many battles on our behalf—thank you Bjarne. Torbjörn Jonsson (who was Bjarne's boss) fought even more battles on Bjarne's behalf—thanks Torbjörn I learnt a lot from you. Jane Walerud—(Jane was the woman behind Open Source Erlang), she was the managing director of Bluetail, and taught me all I know about running a small business.

Now we get to the developers of Erlang which is a much longer list. The original Erlang team was myself, Robert Virding, and Mike Williams. I originally wrote the Erlang compiler, Robert wrote the libraries, and Mike wrote the JAM emulator. Robert and I wrote several different Erlang emulators, mostly in Prolog. Mike re-wrote the emulator in C. After a couple of years Claes Wikström (Klacke) came along and added distribution to Erlang and Bogumil Hausman invented an improved Erlang machine, the BEAM.[1]

[1] Bogdans Erlang Abstract machine.

Many members of the CSLab "dropped into" the project, wrote a few programs in Erlang, then went on do other things. Carl Wilhelm Wellin wrote `yecc`. Klacke and Hans Nilsson wrote `mnesia`, and `mnemosyne`, and Tony Rogvall and Klacke added binaries to the language, and generally did amazing things with networks. Per Hedeland with amazing patience answered all my stupid question about Unix, and made sure our systems always worked beautifully. He also re-wrote the tricky bits in the Erlang emulator when nobody was looking. Magnus Fröberg wrote the debugger, and Torbjörn Törnkvist wrote the interface generator, so that you could interface Erlang with C.

When Erlang moved out of the lab and OTP was born, the group extended and reshaped. Magnus Fröberg, Martin Björklund and I designed the OTP library structure and structure of the behaviours. The OTP behaviours were based on a number of ideas that had been floating around the lab. Klacke had written a "generic server" similar to `gen_server`, and Peter Högfelt had written a generic server, and a early version of the supervision tree. Many of the ideas about process supervision came from Peter's work in the mobility server project.

After I left Ericsson, the day-to-day maintenance, and development of the system moved to a new generation of programmers. Björn Gustavsson maintains the Emulator, and the OTP libraries are maintained by Lars Thorsén, Kenneth Lundin, Kent Boortz, Raimo Niskanen, Patrik Nyblom, Hans Bolinder, Richard Green, Håkan Mattsson, and Dan Gudmundsson.

Now to our users—the Erlang/OTP system has been significantly improved by interaction with our faithful band of users.

The first set of users, who built the first major product in Erlang were, Mats Persson, Kerstin Ödling, Peter Högfeld, Åke Rosberg, Håkan Karlsson, and Håkan, Larsson.

In the AXD301 Ulf Wiger, Staffan Blau, did magnificent work pioneering the use of Erlang for carrier-class applications.

Both inside Ericsson, and outside Ericsson, our users did amazing things. Sean Hinde in the UK became a one-man Erlang factory inside "one-2-one" (now T-mobile).

Almost finally the Erlang mailing list has been a source of inspiration and encouragement. Today if anybody wants to know anything about Er-

lang they just “ask the Erlang list,” and usually get a accurate and informed reply within a hour or so. Thanks to all the people on the list and especially to those who I have never met, but with whom I have exchanged many many long and interesting e-mails.

Finally thanks to my friends and colleagues at SICS—to Prof. Seif Haridi for supervising this thesis. To Per Brand for encouraging me to write the thesis, and for all the other members of the Distributed Systems Laboratory with whom I have had many stimulating discussions.

Thanks everybody.

Programming Rules and Conventions

Program Development Using Erlang Programming Rules and Conventions.[1]

K Eriksson, M Williams, J Armstrong
13 March 1996

Abstract

This is a description of programming rules and advice for how to write systems using Erlang.

Note

This document is a preliminary document and is not complete.

The requirements for the use of EBC's "Base System" are not documented here, but must be followed at a very early design phase if the "Base System" is to be used. These requirements are documented in 1/10268-AND 10406 Uen "MAP - Start and Error Recovery."

[1]This is a reformatted version of the Ericsson internal Document: EPK/NP 95:035—The document was released into the public domain as part of the Open Source Erlang distribution

Contents

1 Purpose

This paper lists some aspects which should be taken into consideration when specifying and programming software systems using Erlang. It does not attempt to give a complete description of general specification and design activities which are independent of the use of Erlang.

2 Structure and Erlang Terminology

Erlang systems are divided into **modules**. Modules are composed of **functions** and **attributes**. Functions are either only visible inside a module or they are **exported** i.e. they can also be called by other functions in other modules. Attributes begin with "-" and are placed in the beginning of a module.

The *work* in a system designed using Erlang is done by **processes**. A process is a *job* which can use functions in many modules. Processes communicate with each other by **sending messages**. Processes **receive** messages which are sent to them, a process can decide which messages it is prepared to receive. Other messages are queued until the receiving process is prepared to receive them.

A process can supervise the existence of another process by setting up a **link** to it. When a process terminates, it automatically sends **exit signals** to the process to which it is linked. The default behaviour of a process receiving an exit signal is to terminate and to propagate the signal to its linked processes. A process can change this default behaviour by **trapping exits**, this causes all exit signals sent to a process to be turned into messages.

A **pure function** is a function that returns the same value given the same arguments regardless of the context of the call of the function. This is what we normally expect from a mathematical function. A function that is not pure is said to have **side effects**.

Side effects typically occur if a function a) sends a message b) receives a message c) calls `exit` d) calls any BIF which changes a process's environment or mode of operation (e.g. get1, put2, erase1, process_flag2 etc).

Warning: This document contains examples of bad code.

3 SW Engineering Principles

3.1 Export as few functions as possible from a module

Modules are the basic code structuring entity in Erlang. A module can contain a large number of functions but only functions which are included in the export list of the module can be called from outside the module.

Seen from the outside, the complexity of a module depends upon the number of functions which are exported from the module. A module which exports one or two functions is usually easier to understand than a module which exports dozens of functions.

Modules where the ratio of exported/non-exported functions is low

are desirable in that a user of the module only needs to understand the functionality of the functions which are exported from the module.

In addition, the writer or maintainer of the code in the module can change the internal structure of the module in any appropriate manner provided the external interface remains unchanged.

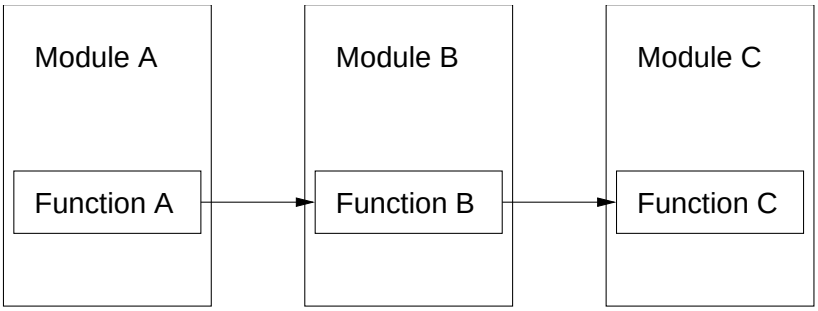
3.2 Try to reduce intermodule dependencies

A module which calls functions in many different modules will be more difficult to maintain than a module which only calls functions in a few different modules.

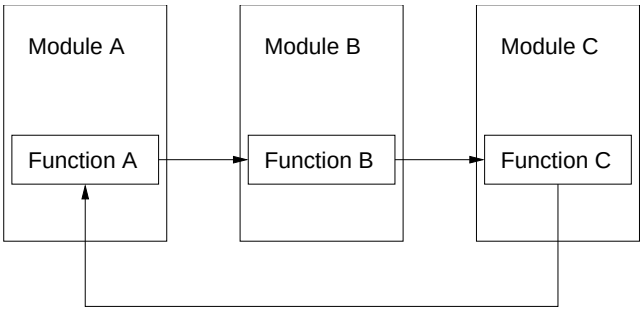
This is because each time we make a change to a module interface, we have to check all places in the code where this module is called. Reducing the interdependencies between modules simplifies the problem of maintaining these modules.

We can simplify the system structure by reducing the number of different modules which are called from a given module.

Note also that it is desirable that the inter-module calling dependencies form a tree and not a cyclic graph. Example:



But not



3.3 Put commonly used code into libraries

Commonly used code should be placed into libraries. The libraries should be collections of related functions. Great effort should be made in ensuring that libraries contain functions of the same type. Thus a library such as `lists` containing only functions for manipulating lists is a good choice, whereas a library, `lists_and_maths` containing a combination of functions for manipulating lists and for mathematics is a very bad choice.

The best library functions have no side effects. Libraries with functions with side effects limit the re-usability.

3.4 Isolate "tricky" or "dirty" code into separate modules

Often a problem can be solved by using a mixture of clean and dirty code. Separate the clean and dirty code into separate modules.

Dirty code is code that does dirty things. Example:

- Uses the process dictionary.
- Uses `erlang:process_info/1` for strange purposes.
- Does anything that you are not supposed to do (but have to do).

Concentrate on trying to maximize the amount of clean code and minimize the amount of dirty code. Isolate the dirty code and clearly comment or otherwise document all side effects and problems associated with this part of the code.

3.5 Don't make assumptions about what the caller will do with the results of a function

Don't make assumptions about why a function has been called or about what the caller of a function wishes to do with the results.

For example, suppose we call a routine with certain arguments which may be invalid. The implementer of the routine should not make any assumptions about what the caller of the function wishes to happen when the arguments are invalid.

Thus we should not write code like:

```
do_something(Args) ->
  case check_args(Args) of
    ok ->
      {ok, do_it(Args)};
    {error, What} ->
      String = format_the_error(What),
      %% Don't do this
      io:format("* error:~s\n", [String]),
      error
  end.
```

Instead write something like:

```
do_something(Args) ->
  case check_args(Args) of
    ok ->
      {ok, do_it(Args)};
    {error, What} ->
      {error, What}
  end.

error_report({error, What}) ->
  format_the_error(What).
```

In the former case the error string is always printed on standard output, in the latter case an error descriptor is returned to the application. The application can now decide what to do with this error descriptor.

By calling `error_report/1` the application can convert the error descriptor to a printable string and print it if so required. But this may not be the desired behaviour - in any case the decision as to what to do with the result is left to the caller.

3.6 Abstract out common patterns of code or behaviour

Whenever you have the same pattern of code in two or more places in the code try to isolate this in a common function and call this function instead of having the code in two different places. Copied code requires much effort to maintain.

If you see similar patterns of code (i.e. almost identical) in two or more places in the code it is worth taking some time to see if one cannot change the problem slightly to make the different cases the same and then write a small amount of additional code to describe the differences between the two.

Avoid “copy” and “paste” programming, use functions!

3.7 Top-down

Write your program using the top-down fashion, not bottom-up (starting with details). Top-down is a nice way of successively approaching details of the implementation, ending up with defining primitive functions. The code will be independent of representation since the representation is not known when the higher levels of code are designed.

3.8 Don’t optimize code

Don’t optimize your code at the first stage. First make it right, then (if necessary) make it fast (while keeping it right).

3.9 Use the principle of “least astonishment”

The system should always respond in a manner which causes the “least astonishment” to the user - i.e. a user should be able to predict what will happen when they do something and not be astonished by the result.

This has to do with consistency, a consistent system where different modules do things in a similar manner will be much easier to understand than a system where each module does things in a different manner.

If you get astonished by what a function does, either your function solves the wrong problem or it has a wrong name.

3.10 Try to eliminate side effects

Erlang has several primitives which have side effects. Functions which use these *cannot be easily re-used* since they cause permanent changes to their environment and you have to know the exact state of the process before calling such routines.

Write as much as possible of the code with side-effect free code.

Maximize the number of pure functions.

Collect together the functions which have side effects and clearly document all the side effects.

With a little care most code can be written in a side-effect free manner - this will make the system a lot easier to maintain, test and understand.

3.11 Don't allow private data structure to "leak" out of a module

This is best illustrated by a simple example. We define a simple module called queue - to implement queues:

```
-module(queue).
-export([add/2, fetch/1]).

add(Item, Q) ->
  lists:append(Q, [Item]).

fetch([H|T]) ->
  {ok, H, T};
fetch([]) ->
  empty.
```

This implements a queue as a list, but unfortunately to use this the user must know that the queue is represented as a list. A typical program to use this might contain the following code fragment:

```
NewQ = [], % Don't do this
Queue1 = queue:add(joe, NewQ),
Queue2 = queue:add(mike, Queue1), ....
```

This is bad - since the user a) needs to know that the queue is represented as a list and b) the implementer cannot change the internal representation of the queue (they might want to do this later to provide a better version of the module).

Better is:

```
-module(queue).
-export([new/0, add/2, fetch/1]).

new() ->
  [].

add(Item, Q) ->
  lists:append(Q, [Item]).

fetch([H|T]) ->
  {ok, H, T};
fetch([]) ->
  empty.
```

Now we can write:

```
NewQ = queue:new(),
Queue1 = queue:add(joe, NewQ),
Queue2 = queue:add(mike, Queue1), ...
```

Which is much better and corrects this problem. Now suppose the user needs to know the length of the queue, they might be tempted to write:

```
Len = length(Queue) % Don't do this
```

since they know that the queue is represented as a list. This is bad programming practice which leads to code which is very difficult to maintain and understand. If they need to know the length of the queue then a length function must be added to the module, thus:

```
-module(queue).
-export([new/0, add/2, fetch/1, len/1]).

new() -> [].

add(Item, Q) ->
  lists:append(Q, [Item]).

fetch([H|T]) ->
  {ok, H, T};

fetch([]) ->
  empty.

len(Q) ->
  length(Q).
```

Now the user can call `queue:len(Queue)` instead.

Here we say that we have "abstracted out" all the details of the queue (the queue is in fact what is called an "abstract data type").

Why do we go to all this trouble? The practice of abstracting out internal details of the implementation allows us to change the implementation without changing the code of the modules which call the functions in the module we have changed. So, for example, a better implementation of the queue is as follows:

```
-module(queue).
-export([new/0, add/2, fetch/1, len/1]).

new() ->
  {[],[]}.

add(Item, {X,Y}) -> % Faster addition of elements
  {[Item|X], Y}.
```

```
fetch({X, [H|T]}) ->
  {ok, H, {X,T}};

fetch({[], []) ->
  empty;

fetch({X, []) ->
  % Perform this heavy computation only sometimes.
  fetch({[],lists:reverse(X)}).

len({X,Y}) ->
  length(X) + length(Y).
```

3.12 Make code as deterministic as possible

A deterministic program is one which will always run in the same manner no matter how many times the program is run. A non-deterministic program may deliver different results each time it is run. For debugging purposes it is a good idea to make things as deterministic as possible. This helps make errors reproducible.

For example, suppose one process has to start five parallel processes and then check that they have started correctly, suppose further that the order in which these five are started does not matter.

We could then choose to either start all five in parallel and then check that they have all started correctly but it would be better to start them one at a time and check that each one has started correctly before starting the next one.

3.13 Do not program "defensively"

A defensive program is one where the programmer does not "trust" the input data to the part of the system they are programming. In general one should not test input data to functions for correctness. Most of the code in the system should be written with the assumption that the input data to the function in question is correct. Only a small part of the code should

actually perform any checking of the data. This is usually done when data "enters" the system for the first time, so once data has been checked as it enters the system it should thereafter be assumed correct.

Example:

```
%% Args: Option is all | normal
get_server_usage_info(Option, AsciiPid) ->
  Pid = list_to_pid(AsciiPid),
  case Option of
    all -> get_all_info(Pid);
    normal -> get_normal_info(Pid)
  end.
```

The function will crash if `Option` neither `normal` nor `all`, and it should do that. The caller is responsible for supplying correct input.

3.14 Isolate hardware interfaces with a device driver

Hardware should be isolated from the system through the use of device drivers. The device drivers should implement hardware interfaces which make the hardware appear as if they were Erlang processes. Hardware should be made to look and behave like normal Erlang processes. Hardware should appear to receive and send normal Erlang messages and should respond in the conventional manner when errors occur.

3.15 Do and undo things in the same function

Suppose we have a program which opens a file, does something with it and closes it later. This should be coded as:

```
do_something_with(File) ->
  case file:open(File, read) of,
    {ok, Stream} ->
      doit(Stream),
      file:close(Stream) % The correct solution
    Error -> Error
  end.
```

Note how we open the file (`file:open`)and close it (`file:close`) in the same routine. The solution below is much harder to follow and it is not obvious which file is closed. Don't program it like this:

```
do_something_with(File) ->
  case file:open(File, read) of,
    {ok, Stream} ->
      doit(Stream)
    Error -> Error
  end.

doit(Stream) ->
  ....,
  func234(...,Stream,...).
  ...

func234(..., Stream, ...) ->
  ...,
  file:close(Stream) %% Don't do this
```

4 Error Handling

4.1 Separate error handling and normal case code

Don't clutter code for the "normal case" with code designed to handle exceptions. As far as possible you should only program the normal case. If the code for the normal case fails, your process should report the error and crash as soon as possible. Don't try to fix up the error and continue. The error should be handled in a different process. (See "Each process should only have one role" on page 229).

Clean separation of error recovery code and normal case code should greatly simplify the overall system design.

The error logs which are generated when a software or hardware error is detected will be used at a later stage to diagnose and correct the error. A permanent record should be kept of any information that will be helpful in this process.

4.2 Identify the error kernel

One of the basic elements of system design is identifying which part of the system has to be correct and which part of the system does not have to be correct.

In conventional operating system design the kernel of the system is assumed to be, and must be, correct, whereas all user application programs do not necessarily have to be correct. If a user application program fails this will only concern the application where the failure occurred but should not affect the integrity of the system as a whole.

The first part of the system design must be to identify that part of the system which must be correct; we call this the error kernel. Often the error kernel has some kind of real-time memory resident data base which stores the state of the hardware.

5 Processes, Servers and Messages

5.1 Implement a process in one module

Code for implementing a single process should be contained in one module. A process can call functions in any library routines but the code for the "top loop" of the process should be contained in a single module. The code for the top loop of a process should not be split into several modules - this would make the flow of control extremely difficult to understand. This does not mean that one should not make use of generic server libraries, these are for helping structuring the control flow.

Conversely, code for no more than one kind of process should be implemented in a single module. Modules containing code for several different processes can be extremely difficult to understand. The code for each individual process should be broken out into a separate module.

5.2 Use processes for structuring the system

Processes are the basic system structuring elements. But don't use processes and message passing when a function call can be used instead.

5.3 Registered processes

Registered processes should be registered with the same name as the module. This makes it easy to find the code for a process.

Only register processes that should live a long time.

5.4 Assign exactly one parallel process to each true concurrent activity in the system

When deciding whether to implement things using sequential or parallel processes then the structure implied by the intrinsic structure of the problem should be used. The main rule is:

"Use one parallel process to model each truly concurrent activity in the real world."

If there is a one-to-one mapping between the number of parallel processes and the number of truly parallel activities in the real world, the program will be easy to understand.

5.5 Each process should only have one "role"

Processes can have different roles in the system, for example in the client-server model.

As far as possible a process should only have one role, i.e. it can be a client or a server but should not combine these roles.

Other roles which processes might have are:

Supervisor watches other processes and restarts them if they fail.

Worker a normal work process (can have errors).

Trusted Worker not allowed to have errors.

5.6 Use generic functions for servers and protocol handlers wherever possible

In many circumstances it is a good idea to use generic server programs such as the `generic` server implemented in the standard libraries. Con-

sistent use of a small set of generic servers will greatly simplify the total system structure.

The same is possible for most of the protocol handling software in the system.

5.7 Tag messages

All messages should be tagged. This makes the order in the receive statement less important and the implementation of new messages easier.

Don't program like this:

```
loop(State) ->
  receive
    ...
    {Mod, Funcs, Args} -> % Don't do this
      apply(Mod, Funcs, Args},
      loop(State);
    ...
  end.
```

The new message `{get_status_info, From, Option}` will introduce a conflict if it is placed below the `{Mod, Func, Args}` message.

If messages are synchronous, the return message should be tagged with a new atom, describing the returned message. Example: if the incoming message is tagged `get_status_info`, the returned message could be tagged `status_info`. One reason for choosing different tags is to make debugging easier.

This is a good solution:

```
loop(State) ->
  receive
    ...
    % Use a tagged message.
    {execute, Mod, Funcs, Args} ->
      apply(Mod, Funcs, Args},
```

```
      loop(State);
    {get_status_info, From, Option} ->
      From ! {status_info,
              get_status_info(Option, State)},
      loop(State);
    ...
  end.
```

5.8 Flush unknown messages

Every server should have an `Other` alternative in at least one `receive` statement. This is to avoid filling up message queues. Example:

```
main_loop() ->
  receive
    {msg1, Msg1} ->
      ...,
      main_loop();
    {msg2, Msg2} ->
      ...,
      main_loop();
    Other -> % Flushes the message queue.
      error_logger:error_msg(
        "Error: Process ~w got unknown msg ~w~n.",
        [self(), Other]),
      main_loop()
  end.
```

5.9 Write tail-recursive servers

All servers *must* be tail-recursive, otherwise the server will consume memory until the system runs out of it.

Don't program like this:

```
loop() ->
  receive
```

```
    {msg1, Msg1} ->
      ...,
      loop();
    stop ->
      true;
    Other ->
      error_logger:log({error, {process_got_other,
                                self(), Other}}),
      loop()
  end,
  % Don't do this!
  % This is NOT tail-recursive
  io:format("Server going down").
```

This is a correct solution:

```
loop() ->
  receive
    {msg1, Msg1} ->
      ...,
      loop();
    stop ->
      io:format("Server going down");
    Other ->
      error_logger:log({error, {process_got_other,
                                self(), Other}}),
      loop()
  end. % This is tail-recursive
```

If you use some kind of server library, for example `generic`, you automatically avoid doing this mistake.

5.10 Interface functions

Use functions for interfaces whenever possible, avoid sending messages directly. Encapsulate message passing into interface functions. There are

cases where you can't do this.

The message protocol is internal information and should be hidden to other modules.

Example of interface function:

```
-module(fileserver).
-export([start/0, stop/0, open_file/1, ...]).

open_file(FileName) ->
  fileserver ! {open_file_request, FileName},
  receive
    {open_file_response, Result} -> Result
  end.

...<code>...
```

5.11 Time-outs

Be careful when using `after` in `receive` statements. Make sure that you handle the case when the message arrives later (See "Flush unknown messages" on page 231).

5.12 Trapping exits

As few processes as possible should trap exit signals. Processes should either trap exits or they should not. It is usually very bad practice for a process to "toggle" trapping exits.

6 Various Erlang Specific Conventions

6.1 Use records as the principle data structure

Use records as the principle data structure. A record is a tagged tuple and was introduced in Erlang version 4.3 and thereafter (see EPK/NP 95:034). It is similar to `struct` in C or `record` in Pascal.

If the record is to be used in several modules, its definition should be placed in a header file (with suffix .hrl) that is included from the modules. If the record is only used from within one module, the definition of the record should be in the beginning of the file where the module is defined.

The record features of Erlang can be used to ensure cross module consistency of data structures and should therefore be used by interface functions when passing data structures between modules.

6.2 Use selectors and constructors

Use selectors and constructors provided by the record feature for managing instances of records. Don't use matching that explicitly assumes that the record is a tuple. Example:

```
demo() ->
  P = #person{name = "Joe", age = 29},
  #person{name = Name1} = P,% Use matching, or...
  Name2 = P#person.name.    % like this.
```

Don't program like this:

```
demo() ->
  P = #person{name = "Joe", age = 29},
  % Don't do this
  {person, Name, _Age, _Phone, _Misc} = P.
```

6.3 Use tagged return values

Use tagged return values.

Don't program like this:

```
keysearch(Key, [{Key, Value}|_Tail]) ->
  Value; %% Don't return untagged values!
keysearch(Key, [{_WrongKey,_WrongValue}|Tail]) ->
  keysearch(Key, Tail);
keysearch(Key, []) ->
  false.
```

Then the Key, Value cannot contain the false value. This is the correct solution:

```
keysearch(Key, [{Key, Value}|_Tail]) ->
  {value, Value}; %% Correct. Returns tagged value.
keysearch(Key, [{_WrongKey, _WrongValue}|Tail]) ->
  keysearch(Key, Tail);
keysearch(Key, []) ->
  false.
```

6.4 Use catch and throw with extreme care

Do not use `catch` and `throw` unless you know exactly what you are doing! Use `catch` and `throw` as little as possible.

`Catch` and `throw` can be useful when the program handles complicated and unreliable input (from the outside world, not from your own reliable program) that may cause errors in many places deeply within the code. One example is a compiler.

6.5 Use the process dictionary with extreme care

Do not use `get` and `put` etc. unless you know exactly what you are doing! Use `get` and `put` etc. as little as possible.

A function that uses the process dictionary can be rewritten by introducing a new argument.

Example:

Don't program like this:

```
tokenize([H|T]) ->
  ...;
tokenize([]) ->
  % Don't use get/1 (like this)
  case get_characters_from_device(get(device)) of
    eof -> [];
    {value, Chars} ->
```

```
        tokenize(Chars)
    end.
```

The correct solution:

```
tokenize(_Device, [H|T]) ->
  ...;
tokenize(Device, []) ->
  % This is better
  case get_characters_from_device(Device) of
    eof -> [];
    {value, Chars} ->
      tokenize(Device, Chars)
  end.
```

The use of `get` and `put` might cause a function to behave differently when called with the same input arguments. This makes the code hard to read since it is non-deterministic. Debugging will be more complicated since a function using `get` and `put` is a function of not only of its input arguments, but also of the process dictionary. Many of the run time errors in Erlang (for example `bad_match`) include the arguments to a function, but never the process dictionary.

6.6 Don't use import

Don't use `-import`, using it makes the code harder to read since you cannot directly see in what module a function is defined. Use `exref` (Cross Reference Tool) to find module dependencies.

6.7 Exporting functions

Make a distinction of why a function is exported. A function can be exported for the following reasons (for example):

- It is a user interface to the module.

- It is an interface function for other modules.

- It is called from `apply`, `spawn` etc. but only from within its module.

Use different `-export` groupings and comment them accordingly. Example:

```
%% user interface
-export([help/0, start/0, stop/0, info/1]).

%% intermodule exports
-export([make_pid/1, make_pid/3]).
-export([process_abbrevs/0, print_info/5]).

%% exports for use within module only
-export([init/1, info_log_impl/1]).
```

7 Specific Lexical and Stylistic Conventions

7.1 Don't write deeply nested code

Nested code is code containing `case/if/receive` statements within other `case/if/receive` statements. It is bad programming style to write deeply nested code - the code has a tendency to drift across the page to the right and soon becomes unreadable. Try to limit most of your code to a maximum of two levels of indentation. This can be achieved by dividing the code into shorter functions.

7.2 Don't write very large modules

A module should not contain more than 400 lines of source code. It is better to have several small modules than one large one.

7.3 Don't write very long functions

Don't write functions with more than 15 to 20 lines of code. Split large functions into several smaller ones. Don't solve the problem by writing long lines.

7.4 Don't write very long lines

Don't write very long lines. A line should not have more than 80 characters. (It will for example fit into an A4 page.)

In Erlang 4.3 and thereafter string constants will be automatically concatenated. Example:

```
io:format("Name: ~s, Age: ~w, Phone: ~w ~n"
   "Dictionary: ~w.~n", [Name, Age, Phone, Dict])
```

7.5 Variable names

Choose meaningful variable names - this is very difficult.

If a variable name consists of several words, use "_" or a capitalized letter to separate them. Example: `My_variable` or `MyVariable`.

Avoid using "_" as don't care variable, use variables beginning with "_" instead. Example: `_Name`. If at a later stage you need the value of the variable, you just remove the leading underscore. You will have no problem finding what underscore to replace and the code will be easier to read.

7.6 Function names

The function name must agree exactly with what the function does. It should return the kind of arguments implied by the function name. It should not surprise the reader. Use conventional names for conventional functions (`start`, `stop`, `init`, `main_loop`).

Functions in different modules that solve the same problem should have the same name. Example: `Module:module_info()`.

Bad function names are one of the most common programming errors - good choice of names is very difficult!

Some kind of naming convention is very useful when writing lots of different functions. For example, the name prefix "`is_`" could be used to signify that the function in question returns the atom true or false.

```
is_...() -> true | false
check_...() -> {ok, ...} | {error, ...}
```

7.7 Module names

Erlang has a flat module structure (i.e. there are no modules within modules). Often, however, we might like to simulate the effect of a hierarchical module structure. This can be done with sets of related modules having the same module prefix.

If, for example, an ISDN handler is implemented using five different and related modules. These module should be given names such as:

```
isdn_init
isdn_partb
isdn_...
```

7.8 Format programs in a consistent manner

A consistent programming style will help you, and other people, to understand your code. Different people have different styles concerning indentation, usage of spaces etc.

For example you might like to write tuples with a single comma between the elements:

```
{12,23,45}
```

Other people might use a comma followed by a blank:

```
{12, 23, 45}
```

Once you have adopted style - stick to it.

Within a larger project, the same style should be used in all parts.

8 Documenting Code

8.1 Attribute code

You must always correctly attribute all code in the module header. Say where all ideas contributing to the module came from - if your code was derived from some other code say where you got this code from and who wrote it.

Never steal code - stealing code is taking code from some other module editing it and *forgetting* to say who wrote the original.

Examples of useful attributes are:

```
-revision('Revision: 1.14 ').
-created('Date: 1995/01/01 11:21:11 ').
-created_by('eklas@erlang').
-modified('Date: 1995/01/05 13:04:07 ').
-modified_by('mbj@erlang').
```

8.2 Provide references in the code to the specifications

Provide cross references in the code to any documents relevant to the understanding of the code. For example, if the code implements some communication protocol or hardware interface give an exact reference with document and page number to the documents that were used to write the code.

8.3 Document all the errors

All errors should be listed together with an English description of what they mean in a separate document (See “Error Messages” on page 246.)

By errors we mean errors which have been detected by the system.

At a point in your program where you detect a logical error call the error logger thus:

```
error_logger:error_msg(Format,
                       {Descriptor, Arg1, ....})
```

And make sure that the line `{Descriptor, Arg1,...}` is added to the error message documents.

8.4 Document all the principle data structures in messages

Use tagged tuples as the principle data structure when sending messages between different parts of the system.

The record features of Erlang (introduced in Erlang versions 4.3 and thereafter) can be used to ensure cross module consistency of data structures.

An English description of all these data structure should be documented (See “Message Descriptions” on page 246.)

8.5 Comments

Comments should be clear and concise and avoid unnecessary wordiness. Make sure that comments are kept up to date with the code. Check that comments add to the understanding of the code. Comments should be written in English.

Comments about the module should not be indented and should start with three percent characters (%%%), (See “File Header, description” on page 243).

Comments about a function should not be indented and start with two percent characters (%%), (See “Comment each function” on page 242).

Comments within Erlang code should start with one percent character (%). If a line only contains a comment, it should be indented as Erlang code. This kind of comment should be placed above the statement it refers to. If the comment can be placed on the same line as the statement, this is preferred.

```
%% Comment about function
some_useful_functions(UsefulArgugument) ->
  another_functions(UsefulArgugument),    % Comment at end of line
  % Comment about complicated_stmnt at the same level of indentation
  complicated_stmnt,
......
```

8.6 Comment each function

The important things to document are:

- The purpose of the function.
- The domain of valid inputs to the function. That is, data structures of the arguments to the functions together with their meaning.
- The domain of the output of the function. That is, all possible data structures of the return value together with their meaning.
- If the function implements a complicated algorithm, describe it.
- The possible causes of failure and exit signals which may be generated by `exit/1`, `throw/1` or any non-obvious run time errors. Note the difference between failure and returning an error.
- Any side effect of the function.

Example:

```
%%--------------------------------------------------------------------
%% Function: get_server_statistics/2
%% Purpose: Get various information from a process.
%% Args:   Option is normal|all.
%% Returns: A list of {Key, Value}
%%     or {error, Reason} (if the process is dead)
%%--------------------------------------------------------------------
get_server_statistics(Option, Pid) when pid(Pid) ->
  ......
```

8.7 Data structures

The record should be defined together with a plain text description. Example:

```
%% File: my_data_structures.h

%%--------------------------------------------------------------------
%% Data Type: person
%% where:
%%    name: A string (default is undefined).
%%    age: An integer (default is undefined).
%%    phone: A list of integers (default is []).
%%    dict:     A dictionary containing various information about the person.
%%       A {Key, Value} list (default is the empty list).
%%--------------------------------------------------------------------
-record(person, {name, age, phone = [], dict = []}).
```

8.8 File headers, copyright

Each file of source code must start with copyright information, for example:

```
%%%----------------------------------------------------------------------
%%% Copyright Ericsson Telecom AB 1996
%%%
%%% All rights reserved. No part of this computer programs(s) may be
%%% used, reproduced,stored in any retrieval system, or transmitted,
%%% in any form or by any means, electronic, mechanical, photocopying,
%%% recording, or otherwise without prior written permission of
%%% Ericsson Telecom AB.
%%%----------------------------------------------------------------------
```

8.9 File headers, revision history

Each file of source code must be documented with its revision history which shows who has been working with the files and what they have done to it.

```
%%%----------------------------------------------------------------------
%%% Revision History
%%%----------------------------------------------------------------------
%%% Rev PA1 Date 960230 Author Fred Bloggs (ETXXXXX)
%%% Initial pre release. Functions for adding and deleting foobars
%%% are incomplete
%%%----------------------------------------------------------------------
%%% Rev A Date 960230 Author Johanna Johansson (ETXYYY)
%%% Added functions for adding and deleting foobars and changed
%%% data structures of foobars to allow for the needs of the Baz
%%% signalling system
%%%----------------------------------------------------------------------
```

8.10 File Header, description

Each file must start with a short description of the module contained in the file and a brief description of all exported functions.

```
%%%----------------------------------------------------------------------
%%% Description module foobar_data_manipulation
%%%----------------------------------------------------------------------
%%% Foobars are the basic elements in the Baz signalling system. The
%%% functions below are for manipulating that data of foobars and for
%%% etc etc etc
%%%----------------------------------------------------------------------
%%% Exports
%%%----------------------------------------------------------------------
%%% create_foobar(Parent, Type)
%%%   returns a new foobar object
%%%   etc etc etc
%%%----------------------------------------------------------------------
```

If you know of any weakness, bugs, badly tested features, make a note of them in a special comment, don't try to hide them. If any part of the module is incomplete, add a special comment. Add comments about anything which will be of help to future maintainers of the module. If the product of which the module you are writing is a success, it may still be changed and improved in ten years time by someone you may never meet.

8.11 Do not comment out old code - remove it

Add a comment in the revision history to that effect. Remember the source code control system will help you!

8.12 Use a source code control system

All non trivial projects must use a source code control system such as RCS, CVS or Clearcase to keep track of all modules.

9 The Most Common Mistakes

- Writing functions which span many pages (See "Don't write very long functions" on page 238).
- Writing functions with deeply nested ifs receives, cases etc (See "Don't write deeply nested code" on page 237).
- Writing badly typed functions (See "Use tagged return values" on page 234).
- Function names which do not reflect what the functions do (See "Function names" on page 238).
- Variable names which are meaningless (See "Variable names" on page 238).

- Using processes when they are not needed (See "Assign exactly one parallel process to each true concurrent activity in the system" on page 229).
- Badly chosen data structures (Bad representations).
- Bad comments or no comments at all (always document arguments and return value).
- Unindented code.
- Using put/get (See "Use the process dictionary with extreme care" on page 235).
- No control of the message queues (See "Flush unknown messages" on page 231 and "Time-outs" on page 233).

10 Required Documents

This section describes some of the system level documents which are necessary for designing and maintaining system programmed using Erlang.

10.1 Module Descriptions

One chapter per module. Contains description of each module, and all exported functions as follows:

- the meaning and data structures of the arguments to the functions
- the meaning and data structure of the return value
- the purpose of the function
- the possible causes of failure and exit signals which may be generated by explicit calls to `exit/1`.

Format of document to be defined later:

10.2 Message Descriptions

The format of all inter-process messages except those defined inside one module.

Format of document to be defined later:

10.3 Process

Description of all registered servers in the system and their interface and purpose.

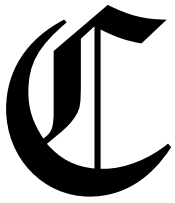
Description of the dynamic processes and their interfaces.

Format of document to be defined later:

10.4 Error Messages

Description of error messages

Format of document to be defined later:



UBF

Getting Erlang to talk to the outside world.[1]

Joe Armstrong
7 May 2002

Abstract

How should Erlang talk to the outside world? — This question becomes interesting if we want to build distributed applications where Erlang is one of a number of communicating components.

We assume these components interact by exchanging messages - at this level of abstraction, details of programming language, operating system and host architecture are irrelevant. What is important is the ease with which we can construct such systems, and the precision with which we can isolate faulty components in such a system. Also of importance is the efficiency (both in terms of CPU and bandwidth requirements) with which we can send and receive messages in the system.

One widely adopted solution to this problem involves the XML family of standards (XML, XML-schemas, SOAP and WSDL) - we argue that

[1]This is a reformatted and slightly revised version of the the paper presented at the ACM SIGPLAN Erlang Workshop–2002 Pittsburg, PA USA[13].

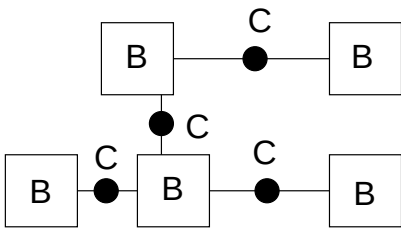


Figure C.1: Black boxes and Contract Checkers

this is inefficient and overly complex and propose basing our system on a simpler binary scheme called UBF (Universal Binary Format). The UBF scheme has the expressive power of the XML set of standards - but is considerably simpler.

UBF has been prototyped in Erlang - the entire scheme (equivalent in semantic power to the XML series of standards) was implemented in a mere 1100 lines of Erlang. UBF encoding of terms is also shown to be more space efficient than the existing "Erlang term format". For example, UBF encoded parse trees of Erlang programs are on average about 60% of the size of the equivalent ETS format encoding which is used in the open source Erlang distribution.

Categories and Subject Descriptors

C.2.2 [**Computer Communications**]: Network Protocols; D.1 [**Software**]: Programming Techniques; D.1 [**Software**]: Programming Languages

1 Introduction

We are interested in building reliable distributed systems out of asynchronously communicating components. We assume that the components are written in different programming languages, run on different operating systems and operate anywhere in the network. For example, some components may be written in Erlang, others in Java, others in C; the components might run on Unix, or Windows or Solaris.

We ask the questions “How should such systems interact?” and “Can we create a convenient language-neutral transport layer to allow such applications to be easily constructed?”

Suppose further that we have several different components and that they collaborate to solve some problem - each individual component has been tested and is assumed to work, and yet the system as a whole does not work. Which component is in error?

There are a number of conventional methods for solving parts of this problem, for example, we could use an interface description language (like Sun XDR [62] or ASN.1 [44]) or we could use a more complex framework like Corba [56]. All these methods have associated problems - many of these methods are supposedly language neutral but in practice are heavily biased to languages like C or C++ and to 32-bit word length processor architectures. The more complex frameworks (like Corba) are difficult to implement and are inappropriate for simple applications. Proprietary solutions for component interaction (like Microsoft’s COM and DCOM) are not considered, since they are both complex and, more damagingly, not available on all platforms.

Out of this mess a single universal panacea has emerged - XML. The XML series of standards, notably XML[20], XML-schemas[32], [65] with SOAP[54], [39], [40] and WSDL[25] has emerged as the universal solution to this problem.

The XML solution involves three layers:

- *A transport layer* - XML provides a simple transport layer. XML encoded terms can be used to encode complex structured objects.
- *A type system* - XML schemas provides a type schema for describing the type of the content of a particular XML tag.
- *A protocol description language* - SOAP defines how simple remote procedure calls can be encoded as XML terms. More complex interactions between components can be described using the Web Service Description Language (WSDL).

The above architectural layering is desirable, in so much as it separates transport of data (XML), the types of the data (XML-schemas) and

the semantics of interactions between different components in the network (SOAP and WDSL).

Unfortunately, while the architecture is essentially correct, the details leave much to be desired. The individual components suffer from a number of significant problems.

We argue in the next section of the paper that XML is overly complex and overly verbose. Following this section we propose a simpler and more efficient but equally expressive binary format, which could be used as a complement to XML.

Our proposed schema has been implemented fully in Erlang and partially in Java and C - we present some preliminary results in the final section of the paper.

Our type system has an expressive power similar to that of the expressive power of XML-schemas, though we believe our scheme to be much simpler. Our contract language has many similarities to WSDL but again we believe it to be simpler and more expressive.

Our architecture also has many similarities to the `.NET` architecture, though we believe our architecture to be simpler and more powerful.

The remainder of the paper describes the system in detail, gives some performance figures and describes our initial experience with the system.

2 Problems with XML

2.1 Complexity

XML, XML-schemas, SOAP and WDSL are a complex set of inter-related standards. A full implementation of the above standards requires many tens of thousands of lines of code and the implementation of a number of minor standards (like XML-name-spaces and XML-path) etc.

The XML standard itself has a grammar of 89 productions and requires many pages of explanatory text - entire text books have been written just to explain the (simple) standard. Having implemented three XML parsers in Erlang I am in the position to say that XML is decidedly not simple to implement - amazingly, most of the complexity occurs in the implementation of a number of features which the vast majority of programmers will

never use (these are antediluvian hang-backs to SGML).

The original design of XML had a notion of structure (described by a regular grammar) but no notion of type. Structure was described using DTDs (Data Type Descriptions) - but the DTD's did not themselves have an XML syntax. This was viewed by some as a disadvantage - XML-schemas came to the rescue - using XML-schemas XML structures could be described in XML itself, and a rich set of types was added.

What was been described by the XML DTD[26]

```
<!ELEMENT ROOT (A?,B+,C*)>
```

became in XML-schemas:

```
<element name="ROOT">
    <complexType content="elementOnly">
        <choice>
           <element ref="t:A">
           <sequence>
               <element ref="t:B">
               <element ref="t:C">
           </sequence>
        </choice>
    </complexType>
</element>
```

The notation for saying that the content of a tag should be of a particular type is equally verbose.

XML-schemas has 19 built-in (or primitive) types and three type constructors The net result of this is that, if you want to express types you have to use XML-schemas. Unfortunately, the verbosity of the specification makes the schemas difficult to read.

In retrospect, a much simpler alternative would have been to extend XML with a small number of primitive data types.

For example, XML has a construct like:

```
<!ELEMENT xxx (#PCDATA)>
```

it would have been easy to extend this with expressions like:

```
<!ELEMENT xxx (#INTEGER32)>
```

Meaning that xxx is a 32 bit integer.

Such an extension would have provided a succinct and readable alternative to XML schemas.

2.2 Verbosity

XML encodings are *incredibly* verbose. The designers of XML excuse themselves with the words: "Terseness in XML markup is of minimal importance."[20] Unfortunately, the very verbosity of XML makes efficient parsing impossible, since at the very least the parser must examine every single input character. This property limits the usefulness of XML as a transport format for mobile devices with limited bandwidth.

Interestingly, one of the most common XML applications designed for such devices, namely WAP, uses an ad hoc method [50] to compress XML WAP programs, providing striking evidence that raw XML is inappropriate as a universal format for low-bandwidth devices.

Another strange property of XML is that binary data must be encoded prior to transmission. For example, a JPEG image must first be base64 encoded. Base64 encoding processes data in 24 bit groups, replacing each 3 byte sequence on input with a 4 byte sequence on output, lines are limited to 76 characters and only printable characters are transmitted.

This is all very strange and highly inefficient (especially considering that SOAP uses TCP/IP for data transport and TCP/IP itself is designed for efficient transport of binary data) - the bit about 76 characters probably has something to do with punched cards, and the restriction to printable characters has something to do with transmission systems that may only pass seven bits of a byte in a transparent manner.

Unfortunately the weird quoting rules of SGML apply to XML - you might naively think that binary data could be transmitted "as is" - unfortunately you can't just quote binary data in XML - if the binary data just happened to contain a valid XML end tag then chaos would ensue.

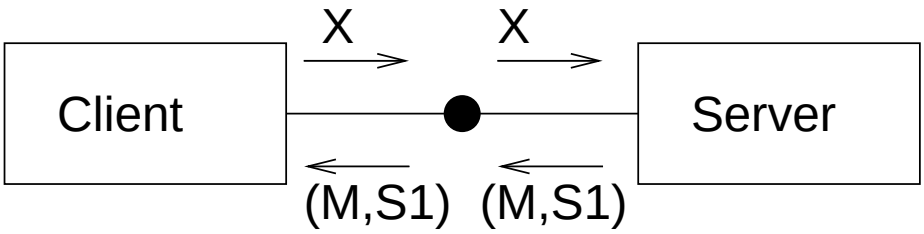


Figure C.2: Client/server with contract checker

Most programming languages have quoting conventions which allow an arbitrary sequence of characters to be quoted, XML does not; thus, for example, any data can be placed within a CDATA block except data containing the string `]]>` - this fact severely limits the usefulness of CDATA section, making it impossible to (say) quote an arbitrary XML program - since it itself might contain a CDATA section. One wonders why such a convention was adopted.

3. Our architecture

Our architecture is shown in Figures D.1 and D.2. Figure D.1 shows a number of communicating components. The components are assumed to be black boxes, at any particular time a component might behave as a client or as a server. Between the components we place an entity which we call a *contract checker* (shown in the diagram as a black blob), the contact checker checks the legality of the flow of messages between the components.

We assume the contract checker starts in state `S` (see Figure D.2); the client sends a message `X` intended for the server, the contract checker checks that `X` is of the correct type and that it is expected in state `S`, if so it is sent to the server. The server responds with a *Message* $\times$ *State* tuple `{M, S1}` the contract checker checks that this message is an expected response in the current state, if so `{M, S1}` is sent to the client and the

state of the contract checker updated to `S1`.[2]

The contract checker is parameterised with a contract that specifies the ordering and types of the allowed message sequences between the client and the server.

The contract is written using a simple non-deterministic finite state machine and a simple type language.

The contract is modeled as a set of four tuples of the form:

$$\{\mathcal{S}_{in}, \mathcal{T}_{in}, \mathcal{T}_{out}, \mathcal{S}_{out}\}$$

This means that if the server is in state $\mathcal{S}_{in}$ and it receives a message of type $\mathcal{T}_{in}$ then it may possibly respond with a message of type $\mathcal{T}_{out}$ and change its state to $\mathcal{S}_{out}$.

The contract checker assumes that the start state of the server is `start` which is assigned to the state variable `S`.

If the client sends the server a message `X` the contract checker checks that there are some rules in the contract where $\mathbf{S} = \mathcal{S}_{in}$ and *typeof*$(\mathbf{X}) = \mathcal{T}_{in}$ - if there are any such rules then the client is said to follow the contract and the message `X` can be safely sent to the server. If no such rules match, then the client is said to have broken the contract and both client and server are informed about this.

If the client has sent a valid message then the set of expected output responses of the server is pruned to a set of two tuples

$$\{\mathcal{T}_{out}, \mathcal{S}_{out}\}$$

being the allowed set of *Type* $\times$ *State* tuples that the server can respond with.

The server must respond with a `{Msg, State}` tuple - the contract manager checks if there is a tuple in the response set where $\mathbf{State} = \mathcal{S}_{out}$ and *typeof*$(\mathbf{Msg}) = \mathcal{T}_{out}$.

If there is such a tuple then the response is accepted and `Msg` is sent back to the client and the global value of the state `S` is updated to `State`.

[2]Note that this is unlike the convention RPC mechanism, where a server responds with a message in response to a particular query, and the next state of the server (if it is statefull) is *implied* by the protocol.

Note that the contract checker operates transparently in normal operation. In the case where both the client and server follow the contract no changes are made to the messages passed between the client and server - the only possible difference between client/server interaction using a contract checker and not using a contract checker is a slight timing difference.

4. UBF - a universal binary format

Contracts are written in a language we call UBF which has two components:

- **UBF(A)** is a data transport format, roughly equivalent to well-formed XML.
- **UBF(B)** is a programming language for describing types in UBF(A) and protocols between clients and servers. UBF(B) is roughly equivalent to verified XML, XML-schemas, SOAP and WDSL.

While the XML series of languages had the goal of having a human readable format the UBF languages take the opposite view and provide a "machine friendly" format.

UBF is designed to be easy to implement. As a proof of concept — UBF drivers for Erlang, Oz, Java and TCL can be found at the author's web site [6]. Implementors are welcome to add new languages.

UBF is designed to be "language neutral" — UBF(A) defines a language neutral binary format for transporting data across a network. UBF(B) is a type system for describing client/server interactions which use UBF(A).

5. UBF(A) - a binary transport format

UBF(A) is a transport format, it is designed to be easy to parse and to be easy to manipulate with a text editor. UBF(A) is based on a byte encoded virtual machine, 26 byte codes are reserved. Instead of allocating the byte codes from 0 we use the printable character codes to make the format easy to read and edit.

Simplicity is the goal, so we define a minimal set of primitive types (four, compared with XML-schemas which have 19) and two types of "glue" for building complex types from more simple types.

5.1 Primitive types

UBF(A) has four primitive types. When a primitive tag is recognized it is pushed onto the "recognition stack" in our decoder. The primitive types are:

Integers - integers are written as sequences of bytes described by the regular expression `[-][0-9]+`. That is, an optional minus (to denote a negative integer) followed by a sequence of one or more digits. No restrictions are made as to the precision of the integer, precision issues are be dealt with in UBF(B).

Strings — strings are written enclosed in double quotes, thus:

```
"....."
```

Within a string two quoting conventions are observed, `"` must be written `\"` and `\` must be written `\\` - no other quotings are allowed (this is so we can write a double quote *within* a string).

Binary Data — binary data is encoded, thus:

```
Int ~....~
```

First an integer, representing the length of the binary data is encoded, followed by a tilde, the data itself, which must be exactly the length given in the integer, and than a closing tilde. The closing tilde has no significance and is retained for readability. White space can be added between the integer length and the data for readability.

Constants — constants are encoded as strings, only using a single quote instead of a double quote.

Constants are commonly found in symbolic languages like Lisp, Prolog or Erlang. In C they would be represented by hashed strings. The essential property of a constant is that two constants can be compared for equality in constant time.

In addition any item can be followed by a *semantic tag* written ‘ . . . ‘. This tag has no meaning in UBF(A) but might have a meaning in UBF(B). For example:

```
12456 ~....~ ‘jpg‘
```

Represents 12456 bytes of raw data with the semantic tag “jpg.” UBF(A) does not know what “jpg” means - this is passed on to UBF(B) which might know what it means - finally the end application is expected to know what to do with an object of type “jpg,” it might for example know that this represents an image. UBF(A) will just encode the tag, UBF(B) will type check the tag, and the application should be able to understand the tag.

5.2 Compound types

Having defined our four simple types we define two type of “glue” for making compound objects.

Structs — structures are written:

```
{ Obj1 Obj2 ... Objn }
```

Where `Obj1..Objn` are arbitrary UBF(A) objects and the byte codes for `{` and `}` are used to delimit a structure. The decoder and encoder must map UBF(A) objects onto an appropriate representation in the application programming language (for example structs in C, arrays in Java, tuples in Erlang etc.).

Structs are used to represent *Fixed numbers of objects*

Lists —lists are used to represent *variable numbers of objects*. They are written with the syntax:

```
# ObjN & ObjN-1 & ... & Obj2 & Obj1
```

This represents a list of objects - the first object in the list is `Obj1` the second `Obj2` etc.- Note that the objects are presented in reverse order. Lisp programmers will recognize # as an operator that pushes NIL (or end of list) onto the recognition stack and & as an operator that takes the top two items on the recognition stack and replaces them by a list cell.

Finally we need to know when an object has finished. The operator $ signifies *end of object.* When $ is encountered there should be only one item on the recognition stack.

5.3 White space

For convenience, blank, carriage return, line feed, tab and comma are treated as white space. Comments can be included in UBF(A) with the syntax %. . .% the usual quoting convention applies.

5.4 Caching optimizations

So far we have used exactly 26 control characters, namely:

```
%"~'‘{}#&\s\n\t\r,-01234567890
```

This leaves us with 230 unallocated byte codes. These are used as follows: The byte code sequence

```
>C
```

Where `C` is not one of the reserved byte codes or > means store the top of the recognition stack in the register `reg[C]` and pop the recognition stack.

Subsequent reuse of the single character `C` means “push `reg[C]` onto the recognition stack.”

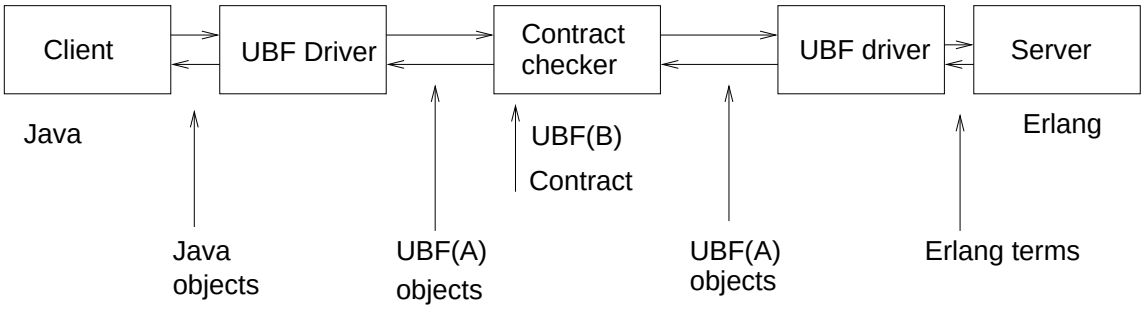


Figure C.3: Client and Server in different languages with a contract checker.

6. Programming by Contract

Central to UBF is the idea of a *contract.* The contract regulates the set of legal conversations that can take place between a client and a server.

A software component (the contract checker) is placed between a client and server and it checks that all interactions between the client and server are legal.

The contract is written using types - the contract says (in a formal language) something like:

> "If I am in state S and you send me a message of type T1 then
> I will reply with a message type T2 and move to state S1, or,
> I will reply with a message of type T3 and move to state S2 ...
> etc."

The contract checker dynamically checks that both sides involved in a transaction obey the contract. Our contracts are expressing in a language we call UBF(B).

UBF(B) has:

A type system - for describing the types of UBF(A) objects.

A protocol description language - for describing client-server interaction in terms of a non-deterministic finite state machine.

An LALR(1) grammar for UBF can be found in appendix A.

6.1 The type system

The type system used here to describe the type of UFB(A) encoded objects is a simplified version of the type system used to describe Erlang terms[8]. The notation:

- `int()` Means a UBF(A) integer.
- `string()` Means a UBF(A) string.
- `constant()` Means a UBF(A) constant.
- `bin()` Means a UBF(A) binary data item.
- `X()` Means an Object of type `X`

UBF(A) literals are written as follows:

- `"..."` - denotes a UBF(A) string.
- `[a-z][a-zA-Z0-9_]*` - denotes a UBF(A) constant.
- `[-][0-9]+` - denotes a UBF(A) integer.

Complex types are defined recursively:

{`T1, T2, ..., Tn`} Is the *tuple type* if `T1` .. `Tn` are types. We say that `{X1, X2, ..., Xn}` is of type `{T1, T2, ..., Tn}` if `X1` is of type `T1`, `X2` is of type `T2`, ... and `Xn` is of type `Tn`.

$[T]$ Is the *list type* if `T` is a type. We say that `# Xn & Xn-1 & ... X2 & X1` is of type `[T]` if all `Xi` are of type `T`.

$T1|T2$ Is the *alternation type* if `T1` and `T2` are types. We say that `X` is of type `T1 | T2` if `X` is of type `T1` or if `X` is of type `T2`.

6.2 New types

New types are introduced in UBF(B) with the notation:

```
+TYPES X() = Type1; Type2; ...
```

Where `Type1`, `Type2`, ... are simple types, literal types or complex types.

Examples of types are:

```
+TYPES
  person()    = {person,
                 firstname(),
                 lastname(),
                 sex(),
                 age()};
  firstname() = string();
  lastname()  = string();
  age()       = int();
  sex()       = male | female;
  people()    = [person()].
```

This type schema defines a number of different types. For example, it is easily seen that:

```
'person' >p
# {p "jim" "smith" 'male' 10} &
  {p "susan" "jones" 'female' 14} & $
```

Is of type `people()`.

Note that unlike XML UBF(A) encoded terms do not contain any tag information. To make this clearer, suppose we had made an XML data structure to represent the same information, this might be something like:

```
<people>
  <person>
```

```
      <firstname>jim</firstname>
      <lastname>smith</lastname>
      <sex>male</sex>
      <age>10</age>
    </person>
    <person>
      <firstname>susan</firstname>
      <lastname>jones</lastname>
      <sex>female</sex>
      <age>14</age>
    </person>
  </people>
```

The XML data structure contains a large number of redundant tags - whereas our representation omits all the tags The sizes of the first representation is 65 bytes and the second 215 (ignoring white space which is redundant) - we might thus expect that parsing the UBF expression would be at least three times as fast as parsing the XML expression.

Note that UBF(B) type is a *language independent type schema*. It defines the types of messages after encoding, and is thus universally applicable to any programming language which produces UBF encoded data.

Language independent type schemas are the basis of *Contracts* between clients are servers.

6.3 The Contract Language

We start with a simple example:

```
+NAME("file_server").
+VSN("ubf1.0").

+TYPES
info()         = info;
description()  = description;
services()     = services;
```

```
contract()     = contract;

file()         = string();
ls()           = ls;
files()        = {files, [file()]};
getFile()      = {get, file()};
noSuchFile()   = noSuchFile.

+STATE start
       ls()      => files()     & start;
       getFile() => binary()    & start
                 | noSuchFile() & stop.

+ANYSTATE
        info()        => string();
        description() => string();
        contract()    => term().
```

The program starts with a sequence of type definitions (they follow the TYPES keyword) - these define the types of the message that are visible across the interface to the component.

Here, for example we see type `getFile()` is defined as `{get,file()}` where `file()` is of type `string()`.

Given this definition it can easily be seen that the UBF(A) sequence of characters `{'get' "image.jpg"}$` belongs to this type.

Reading further (in the STATE part of the program) we see the rule:

```
+STATE start
     ls()      => files()     & start;
     getFile() => bin()       & start
               | noSuchFile() & stop.
```

In English, this rule means:

> If the system is in the state `start` and if it receives a message of type `ls()` then respond with a message of type `files()`

> and move into the `start` state, otherwise, if a message of type `getFile()` is received then either respond with a message of type `bin()` and move to the state `start`, or respond with a message of type `noSuchFile()` and move to the state `stop`.

To continue with our example, we requested a file named `image.jpg` the valid responses are of type `bin()` or `noSuchFile()` which corresponds to UBF(A) encoded sequences like `NNN~ ... ~$` or `'noSuchFile'$`.

Note that it might not always be possible for a component to distinguish between two different state transitions on the basis of the response alone. Consider the following fragment of a contract:

```
+TYPES running() = string();
       error()   = string().

+STATE running
       request() => ok() & running;
                  |  error() & stopping.
```

If we knew a component was in the state `running` and we sent it a message of type `request()` then we would expect it to respond with one of the types `ok()` or `error()` - unfortunately these types are indistinguishable, since both are represented as strings in UBF(A). For this reason we require that the server responds with a `State X Message` pair, not just a message. The server *explicitly* reveals its next state to the contract checker.

7. Implementation details

The entire UBF system has been prototyped in Erlang. The entire system is about 1100 lines of commented Erlang code.

- UBF encoding/decoding 391 lines.
- Contract parsing 270 lines.

- Contract checker and type checker - 301 lines.
- Run-time infrastructure and support libraries - 130 lines.

This compares favourably with the complexity of an XML implementation. As an example an incomplete implementation of XML which I wrote two years ago has 2765 lines of Erlang code. This should be compared with the 391 lines of code in the UBF implementation.

8. Performance

So far, the system has been implemented entirely in Erlang and no thought given to embedding the UBF encoding/decoding software and the type checking software into the Erlang run-time system.

The only measure of performance we give here concerns the packing density of UBF encoded Erlang terms.

As a simple check we compared the size of the encoding of the parse tree of a number of Erlang modules, with the size of the a binary produced by evaluating the expression:

```
size(term_to_binary(epp:parse_file(F, [], [])))
```

The algorithm used to serialize the term representing the parse tree was a simple two-pass optimizer which cached the most frequently used constants which occurred in the program.

Based on a sample of 24 files we observed that on average the UBF(A) encoded data was 59% of the size of the corresponding data when encoded in the Erlang external term format. In applications where bandwidth is expensive and communication relatively slow (for example, communication using mobile devices and GPRS) such a reduction in data volume would have considerable benefit.

9. Future work

Our system of contracts uses only a very simple type system. It is easy to envisage extensions to allow more complex types and extensions to describe non-functional properties of the system.

The non-functional properties of the system are of particular interest. An example of these might be to add simple timing constraints, allowing rules such as:

```
+STATE S1
   T1 => T2 & S2 before Time1
      |  T3 & S3 after  Time2
...
```

meaning that if a component is in state `S1` and receives a message of type `T1` then it might respond with a message of type `T2` and change to state `S2` within `Time1` or, respond with a message of type `T3` and change state to a state `S3` after a time `Time2`.

Stricter contracts with timing constraints could be very useful in designing real-time systems of interacting components.

Other extensions could be imagined which would allow us to define contracts like subroutines - so that one contract could use a sub-contract to perform a specific task.

10. Running the system

Since our system essentially exchanges characters, we can use telnet to observe a session and test the behaviour of the system. Here is an example of commands issued in a telnet session where the client is talking directly to the file server specified by the `file_server` contract given above:

```
'info'$
{"I am a mini file server",'start'}$
```

Recall that the system starts in the state `start` and that the contract says that the `info` command can be sent in any state. The response should be a string, and the new state (in this case `start` since the state is not changed by an `ANYSTATE` rule).

The application returns a two tuple, containing a descriptive string and the new state. This is converted by the application driver to the UBF tuple `{"I am ... ", 'start'}$`.

```
      'ls'$
      {{'files',
        #
        "ubf.erl"&
        "client.erl"&
        "Makefile"& ...}
       'start'}$
```

Here the client sends a message of type `ls()` - the server responds with tuple `{{'files',#..., 'start'}$` message. This first element in the tuple is of type `files()`.

Finally we ask the system to describe itself:

```
'contract'$
{'contract',
 {{'name',"file_server"},
  {'info',"I am a mini file server"},
  {'description',"

Commands:

    'ls'$  List files
    {'get' File} => Length ~ ... ~
                  | noSuchFile
"},
  {'services',#},
  {'states',
   #{'start',
     #{'input',{'tuple',#{'prim','file'}&
        {'constant','get'}&},
     #{'output',{'constant','noSuchFile'},'stop'}&
      {'output',{'prim','binary'},'start'}&}&
     {'input',{'constant','ls'},
     #{'output',
        {'tuple',
```

```
          #{'list',{'prim','string'}}&
        {'constant','files'}&},'start'}&}&}&},
  {'types',
        #{'file',{'prim','string'}}&}}}$
```

The system responds to a message of type `info()` with a parse tree representing the contract itself.

In the contract itself we used the generic type `term()` to describe the contract. The contract itself is a well typed term in UBF, but a discussion of the abstracted form of the contract itself is not relevant to this paper.

The example is given to illustrate the *introspective* power of the system. Not only can we run the system, we can also ask the system to describe itself. We believe this to be a desirable property of a distributed component in a system of communicating components.

11. A larger contract

Our previous examples showed the basic syntax of a contract. We finish with a more complex example. The contract below describes an IRC[46] like protocol.

```
+NAME("irc").

+VSN("ubf1.0").


+TYPES

info()          = info;
description()   = description;
contract()      = contract;

bool()          = true | false;
nick()          = string();
oldnick()       = string();
```

```
newnick()      = string();
group()        = string();
logon()        = logon;
proceed()      = {ok, nick()}
listGroups()   = groups;
groups()       = [group()];
joinGroup()    = {join, group()}
leaveGroup()   = {leave, group()};
ok()           = ok;
changeNick()   = {nick, nick()}
%%               send a message to a group";
msg()          = {msg, group(), string()}
msgEvent()     = {msg, nick(), group(), string()};
joinEvent()    = {joins, nick(), group()};
leaveEvent()   =  {leaves, nick(), group()};
changeNameEvent() = {changesName,
                     oldnick(),newnick(), group()}.

%% I am assigned an initial (random) nick

+STATE start logon() => proceed() & active.

+STATE active

   listGroups() => groups() & active;
   joinGroup()  => ok() & active;
   leaveGroup() => ok() & active;
   changeNick() => bool() & active;
%%                 false if not in group
   msg()        => bool() & active;

   EVENT => msgEvent();        % Message from group
   EVENT => joinEvent();       % Nick joins group
   EVENT => leaveEvent();      % Nick leaves group
   EVENT => changeNameEvent(). % Nick changes name
```

```
+ANYSTATE
        info()        => string();
        description() => string();
        contract()    => term().
```

This example introduces a new keyword `EVENT`. The syntax:

```
+STATE S1
   ...
   EVENT => T2;
   ...
```

means that the server can spontaneously send a message of type `T2` to the client. Normally, messages are sent to the client in response to requests, `EVENT` is used for asynchronous single messages from the server to the client. Since the server cannot be sure that the client has received such a message no change of state in the server is allowed.

12. Experience

The initial version of UBF was completed in about three weeks of intensive programming - the system design changed many times and was re-designed, implemented and re-implemented several times.

Once the basic infrastructure was running, a simple interface to Oz was implemented - and following this an interface to Java. The Oz and Java implementation only concerned UBF(A) and not the contract language or checker.

The first non-toy application (IRC) was implemented to test the system on a non-trivial example. I started by writing the contract and then made an Erlang client and server which followed the contract.

Interestingly the contract checker proved extremely helpful in developing the IRC system - I often develop systems by writing a client and server in the same time frame, shifting attention between the client and server as necessary. Using the contract checker proved helpful in rapidly identifying

which of the two components was in error in the event of an error. Also, since the intermediate communication has a fairly readable ASCII subset I was able to test the server by typing simple text queries in a telnet session - in this way I was able to immediately test the server (and the interaction between the client and server) using telnet, rather than my Erlang code (which at some stages was only partially complete).

Interestingly the contract checker often complained about contract violations that I did not believe, so I erroneously assumed that the code for checking the contracts was incorrect. Almost invariably the contract checker was right and I was wrong. I think we have a tendency to believe what we had expected to see - and not that which was actually present - the contract checker had no such biases.

Concentration on the contact itself caused an interesting psychological shift of perspective and forced me to think about the system in meta-level terms considering the client and server as only stupid black boxes which did what they were told. Trying to write the contracts in a clear manner was also an exercise which resulted in a clearer understanding of the problem by forcing me to think in terms of what messages are sent between the client and server - and nothing else.

The contract proved also a valuable and easy-to-understand specification of the problem. Having implemented an Erlang client and server and a graphic based Erlang client we decided to add a Java client.

The Java client was developed independently by Luke Gorrie using only the UBF specification and the irc contract. When it came to testing the contract checker could provide extremely precise error diagnostics - of the form:

> I was in state $\mathcal{S}$ and I expected you to send me a message of type $\mathcal{T}$ but you sent me the message `M` which is wrong.

Armed with such precise diagnostics it was easy to debug the Java program. Needless to say when the Java client talked to the Erlang server the system worked first time. Testing both the Java client and the Erlang server could be done independently using only a modified form of the contract checker and the contract concerned.

Having developed the system we have a high degree of confidence in its correctness - and if it should fail we'll immediately know which component is broken.

13. Acknowledgments

I would like to thank Seif Haridi, Per Brand, Thomas Arts, and Luke Gorrie for their helpful discussions - particular thanks go to Luke for implementing the Java client.

APPENDIX

UBF grammar

```
form -> '+' 'NAME '(' string ')' dot.
form -> '+' 'VSN' '(' string ')'  dot.
form -> '+' 'TYPES' types dot.
form -> '+' 'STATE' atom transitions dot.
form -> '+' 'ANYSTATE'anyrules dot.

types -> typeDef ';' types.
types -> typeDef.

typeDef -> atom '(' ')' '=' type annotation.

annotation -> string.
annotation -> '$empty'.

type ->  primType '|' type.
type ->  primType.

primType -> 'int'      '(' ')'.
primType -> 'string'   '(' ')'.
primType -> 'constant' '(' ')'.
primType -> 'bin'      '(' ')'.
```

```
primType -> atom       '(' ')'.
primType -> '{' typeSeq '}'.
primType -> '[' type ']'.
primType -> atom.
primType -> integer.
primType -> integer '.' '.' integer.
primType -> string.

typeSeq -> type.
typeSeq -> type ',' typeSeq.

typeRef -> atom '(' ')'.

transitions -> transition ';' transitions.
transitions -> transition.

transition -> typeRef '=>' outputs.
transition -> 'EVENT' '=>' typeRef.

outputs -> responseAndState '|' outputs.
outputs -> responseAndState.

responseAndState -> typeRef '&' atom.

anyrules -> anyrule ';' anyrules.
anyrules -> anyrule.

anyrule -> typeRef '=>' typeRef.

strings -> string ',' strings.
strings -> string.
strings -> '$empty'.
```

Colophon

It had been my intention to write a new typesetting system in order to typeset this thesis—I therefore started work on *Erlguten* [7] which has been described elsewhere.

Having read Knuth's masterwork *digital typography* I realised that producing a high quality type-setting system would delay the work on the thesis by several years (in my estimate about five years) and so I have decided, reluctantly to use LaTeX.

It turns out that with a little tweaking even pdfLaTeX can be persuaded to produce almost readable text.

This thesis was produced for A4 paper in 14 pt FSBaskerville, reduced by 81% to fit onto A5 paper. The drop capitals and chapter numbers are set in Old English and all computer programs, shell dialogs, and references to code in the text are set in Computer Modern.

John Baskerville (1706–1775) was an English calligrapher, stonecutter, type designer, and printer. In 1750 he started a printing business, but being a perfectionist his first work was delayed until 1757. His work was much admired by Fournier, Bodoni, and Benjamin Franklin.

I chose the Baskerville typeface because it has heavily weighted hairlines[1] and thus is well suited for documents that are produced on a low quality laser printer or for photocopying.

Text using typefaces (like Garamond) which have delicate serifs, and lightly weighted hairlines tends to fade when printed with a poor quality printer on poor quality paper. This fading reduces the legibility of the text.

[1] Hairlines are the thinnest lines used in a letterform.

I have also tried using the micro-typographic extensions developed by Hàn Thê Thành in his doctoral thesis [64], which reintroduced the original practice used in the Gutenberg 42 line bible of allowing punctuation at the end of lines to slightly protrude into the right-hand margin. This practice improves the optical alignment of the left-hand margin and produces better line breaks.

Unfortunately I abandoned this approach, since the software for margin kerning did not always work correctly with multiple typefaces on the same line. For this reason alone further work on *Erlguten* seems motivated.

Finally, I would like to thank the small army of people who have read and commented on this thesis. My wife, Helen, proof read the entire thesis and found an embarrassing number of small mistakes in the text. Peter Van Roy, and Ulf Wiger found some spelling mistakes that even Helen had missed and Richard O'Keefe found some formatting errors that everybody missed.

BIBLIOGRAPHY

[1] Ingemar Ahlberg, John-Olof Bauner, and Anders Danne. Prototyping cordless using declarative programming. *XIV International Switching Symposium*, October 1992.

[2] Leon Alkalai and Ann T. Tai. Long-life deep-space applications. *IEEE Computer*, 31:37–38, April 1998.

[3] Marie Alpman. Svenskt internetbolag köps för 1,4 miljarder. *Ny Teknik*, August 2000.

[4] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys (CSUR)*, 15(1):3–43, 1983.

[5] J. Armstrong, M. Williams, C. Wikström, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.

[6] J. L. Armstrong. Ubf - universal binary format, http://www.sics.se/ joe/ubf. 2002.

[7] J. L. Armstrong. Erlguten. 2003. http://www.sics.se/ joe/erlguten.html.

[8] J. L. Armstrong and T. Arts. A practical type system for erlang. *Erlang User Conference*, 2002.

[9] J. L. Armstrong, B. O. Däcker, S. R. Virding, and M. C. Williams. Implementing a functional language for highly parallel real-time applications. In *Software Engineering for Telecommunication Switching Systems*, April 92.

[10] J. L. Armstrong, S. R. Virding, and M. C. Williams. Use of Prolog for Developing a New Programming Language. In C. Moss and

K. Bowen, editors, *Proc. 1st Conf. on The Practical Application of Prolog*, London, England, 1992. Association for Logic Programming.

[11] Joe Armstrong. Increasing the reliability of email services. In *Proceedings of the 2000 ACM symposium on Applied Computing*, pages 627–632. ACM Press, 2000.

[12] Joe Armstrong. Concurrency oriented programming. *Lightweight Languages Workshop (LL2)*, November 2002.

[13] Joe Armstrong. Getting erlang to talk to the outside world. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 64–72. ACM Press, 2002.

[14] Joe Armstrong. Concurrency oriented programming in erlang. *GUUG 2003*, March 2003.

[15] Joe Armstrong. A webserver daemon. 2003. This is available at `http://www.sics.se/~joe/tutorials/web_server/web_server.html`.

[16] A. Avienis. Design of fault-tolerant computers. In *Proceedings of the 1967 Fall Joint Computer Conference. AFIPS Conf. Proc., Vol. 31, Thompson Books, Washington, D.C., 1967, pp. 733-743*, pages 733–743, 1967.

[17] Jonas Barklund. Erlang 5.0 specification. 2000. available from `http//www.bluetail.com/~rv`.

[18] Staffan Blau and Jan Rooth. Axd 301 – a new generation of atm switching. *Ericsson Review*, (1), 1998.

[19] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., 1999.

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler (Eds). Extensible markup language (xml) 1.0 (second edition). october 2000, http://www.w3.org/tr/2000/rec-xml-20001006. 2000.

[21] Ciaràn Bryce and Chrislain Razafimahefa. An approach to safe object sharing. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 367–381. ACM Press, 2000.

[22] George Candea and Armando Fox. Crash only software. In *Proceedings of the 9th workshop on Hot Topics in Operating Systems (TotOS-IX)*, May 2003.

[23] Richard Carlsson, Thomas Lindgren, Björn Gustavsson, Sven-Olof Nyström, Robert Virding, Erik Johansson, and Mikael Pettersson. Core erlang 1.0. November 2001.

[24] J. D. Case, M. S. Fedor, M. L. Schoffstall, and C. Davin. Simple network management protocol (SNMP). RFC 1157, Internet Engineering Task Force, May 1990.

[25] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, march 2001, http://www.w3.org/tr/2001/note-wsdl-20010315/. 2001.

[26] Dan Connolly, Bert Bos, Yuichi Koike, and Mary Holstege. `http://www.w3.org/2000/04/schema_hack/`. 2000.

[27] M. R. Crispin. Internet message access protocol - version 4. RFC 1730, Internet Engineering Task Force, December 1994.

[28] Grzegorz Czajkowski and Laurent Daynès. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 125–138. ACM Press, 2001.

[29] Bjarne Däcker. Datalogilaboratoriet - de första 10 åren. March 1994.

[30] Bjarne Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. November 2000. Licentiate Thesis.

[31] A. Dahlin, M. Froberg, J. Grebeno, J.Walerud, and P. Winroth. Eddie: A robust and scalable internet server. May 1998.

[32] D. C. Fallside (Ed). Xml schema part 0: Primer. may 2002. http://www.w3.org/tr/2001/rec-xmlschema-0-20010502/. 2002.

[33] Dick Eriksson, Mats Persson, and Kerstin Ödling. A switching software architecture prototype using real time declarative language. *XIV International Switching Symposium*, October 1992.

[34] Open source erlang distribution. 1999.

[35] J. A. Feldman, J. R. Low, and P. D. Rovner. Programming distributed systems. In *Proceedings of the 1978 ACM Computer Science Conference*, pages 310–316, 1978.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, The Internet Society, June 1999. See `http://www.ietf.org/rfc/rfc2616.txt`.

[37] Ian Foster and Stephen Taylor. *Strand: new concepts in parallel programming*. Prentice-Hall, Inc., 1990.

[38] Jim Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, 1985.

[39] M. Gudgin, M. Hadley, J-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 1: Messaging framework, december 2001, http://www.w3.org/tr/2001/wd-soap12-part1-20011217. 2001.

[40] M. Gudgin, M. Hadley, J-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 2: Adjuncts, december 2001, http://www.w3.org/tr/2001/wd-soap12-part2-20011217. 2001.

[41] Bogumil Hausman. Turbo erlang. *International Logic Programming Symposium*, October 1993.

[42] Bogumil Hausman. Turbo erlang: Approaching the speed of c. *In Evan Tick and Giancarlo Succi, editors, Implementations of Logic Programming Systems, pages 119–135. Kluwer Academic Publishers*, 1994.

[43] American National Standards Institute, Institute of Electrical, and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[44] ISO/IEC. Osi networking and system aspects - abstract syntax notation one (asn.1). ITU-T Rec. X.680 — ISO/IEC 8824-11, ISO/IEC, 1997.

[45] ITU. Recommendation Z.100 – specification and description language (sdl). ITU-T Z.100, International Telecommunication Union, 1994.

[46] D. Reed J. Oikarinen. RFC 1459: Internet relay chat protocol. May 1993.

[47] Erik Johansson, Sven-Olof Nyström, Mikael Pettersson, and Konstantinos Sagonas. Hipe: High performance erlang.

[48] D. Richard Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer*, 30(4):31–36, 1997.

[49] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *International Conference on Functional Programming*, pages 136–149. ACM, June 1997.

[50] B. Martin and B. Jano (Eds). Wap binary xml content format, june 1999, http://www.w3.org/tr/wbxml. 1999.

[51] Håkan Millroth. Private communication. 2003.

[52] J. Myers and M. P. Rose. Post office protocol - version 3. RFC 1939, Internet Engineering Task Force, May 1996.

[53] Nortel Networks. Alteon ssl accelerator product brief. September 2002.

[54] (Ed) Nilo Mitra. Soap version 1.2 part 0: Primer. december 2001, http://www.w3.org/tr/2001/wd-soap12-part0-20011217. 2001.

[55] Hans Olsson. Ericsson lägger ner utveckling. *Dagens Nyheter*, December 1995.

[56] OMG. *Common Object Request Broker Architecture (CORBA)–v2.6.1 Manual*. The Object Management Group, Needham, U.S.A, 2002.

[57] J. B. Postel. Simple mail transfer protocol. RFC 821, Internet Engineering Task Force, August 1982.

[58] K. Renzel. Error handling for business information systems. 2003.

[59] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3):222–238, 1983.

[60] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems (TOCS)*, 2(2):145–154, 1984.

[61] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[62] R. Srinivasan. RFC 1832: XDR: External data representation standard. August 1995.

[63] Ann T. Tai, Kam S. Tso, Leon Alkalai, Savio N. Chau, and William H. Sanders. On the effectiveness of a message-driven confidence-driven protocol for guarded software upgrading. *Performance Evaluation*, 44(1-4):211–236, 2001.

[64] Hàn Thê Thành. Micro-typographic extensions to the tex typesetting system. *Masaryk University Brno*, 2000.

[65] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (Eds). Xml schema part 1: Structures. w3c recommendation, may 2001. http://www.w3.org/tr/2001/rec-xmlschema-1-20010502/. 2001.

[66] Seved Torstendahl. Open telecom platform. *Ericsson Review*, (1), 1997.

[67] Jeffrey Voas. Fault tolerance. *IEEE Software*, pages 54–57, July–August 2001.

[68] David H.D. Warren. An abstract Prolog instruction set. Tecnical Note 309, SRI International, Menlo Park, California, October 1983.

[69] Ulf Wiger. Private communication.

[70] Ulf Wiger, Gösta Ask, and Kent Boortz. World-class product certification using erlang. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 24–33. ACM Press, 2002.

[71] Weider D. Yu. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, 3(2), 1998.